MASTER EN SCIENCES INFORMATIQUES

Simulation d'un laboratoire d'électricité

Meert, Arnaud

Award date:
1989

Awarding institution:
Universite de Namur

INSTITUT d'INFORMATIQUE

SIMULATION

d'un

LABORATOIRE d'ELECTRICITE

Arnaud **MEERT**

Promoteur : C. CHERTON

Mémoire présenté en vue de l'obtention du
titre de Licencié et Maître en informatique

Année Académique 1988-1989

RESUME

Ce mémoire tente de répondre à une question : est-il justifié de vouloir utiliser un ordinateur dans le cadre d'un cours d'électricité ?
Après avoir précisé le contexte de l'EAO et ce que l'on entend par un laboratoire d'électricité dans l'enseignement secondaire, nous avons montré qu'un certain nombre d'améliorations pouvaient être apportées grâce à la simulation d'un tel laboratoire.
Nous avons ensuite défini le type de circuits qu'il serait possible de simuler à l'aide d'un seul didacticiel et nous avons développé une stratégie de résolution qui permet d'aider à localiser les dysfonctionnements d'un circuit.
A partir de là, il restait à définir un ensemble d'outils "modulaires" et "réutilisables" qui permettent d'apporter un élément de solution au problème particulier qui nous occupe tout en présentant des caractéristiques assez générales pour aider à résoudre d'autres problèmes.
Enfin, à l'aide de ces fragments de solution, nous avons développé le début d'une implémentation d'un simulateur de laboratoire d'électricité.

ABSTRACT

This memoir tries to answer to the question : is it justified to make use of a computer to learn electric-circuit theory ?
First, we have defined the environment of CAL and what is ans electric laboratory in a secondary scholl : we have shown that a lot of improvements can be achieved by simulating such a laboratory.
Then we have determined the type of circuits wich could be simulated with one educational software and we have developed a solving method wich can help to locate circuit mis'function.
Afterwards, it remains to define a set of "modular" and "reusable" tools : they are solving elements for our particular problem and enough general to help in solving another one.
Finally, with these solution pieces, we have developed an embryo of electric laboratory simulator.

Je tiens à remercier Monsieur C. Cherton pour l'aide précieuse qu'il m'a apporté dans l'élaboration de ce mémoire et ce, tout en me laissant un grand espace de liberté...

Je rends également hommage à mon épouse pour les trésors de patience qu'elle a su déployer pendant cette longue tâche...

TABLE des MATIERES

INTRODUCTION

Dans le cadre d'un mémoire traitant d'enseignement assisté par ordinateur, il convient d'abord, me semble-t-il, de situer rapidement le type de relations qui peuvent exister entre l'informatique et l'enseignement.

Si l'on considère ce dernier dans son ensemble, on constate qu'à tous les niveaux, du primaire à l'université, l'ordinateur est très rarement utilisé comme outil pédagogique. Ce phénomène se rencontre également dans le cadre de la formation d'informaticiens bien que dans ce cas l'environnement soit, à maints égards, beaucoup plus favorable que dans d'autres branches.

ORDINATEUR-OUTIL PEDAGOGIQUE ?

La première question qui se pose est de savoir si l'ordinateur est un outil pédagogique de valeur.

Si l'on regarde quels "outils" l'enseignement utilise, on trouve tout d'abord une panoplie de moyens de communication "audio-visuels" (livre, tableau noir,...), ensuite des objets concrets destinés à faire saisir des concepts abstraits (réglette-arithmétique) et enfin un homme-orchestre mettant en oeuvre tout ce matériel dans une optique pédagogique.
Comme on le voit, un outil n'est pas intrinsèquement pédagogique : c'est surtout son utilisation fructueuse dans un contexte d'enseignement qui le fait considérer comme tel.

Dans d'autres secteurs, on observe que l'informatique est utilisée avec fruit dans des techniques que l'on rencontre aussi dans l'enseignement.
Voyons en quelques exemples :

- On retrouve l'ordinateur comme outil de synthèse d'images dans les secteurs audio-visuels. Qu'en est-il dans l'enseignement en ce qui concerne l'illustration de phénomènes dynamiques ?
- Qu'il s'agisse de la construction de voitures ou d'engins spaciaux, on utilise la puissance de l'informatique pour simuler le comportement d'un objet qui n'existe même pas encore physiquement à l'état de prototype. Dans bon nombre de cours, il est impossible de disposer de tous les matériels existants, souvent très couteux et rapidement dépassés. La simulation de ceux-ci serait bien plus évocatrice que leur description.
- Enfin, il existe des tâches répétitives pour lesquelles il faut acquérir un certain nombre d'automatismes (guichet d'une banque, traitement de texte particulier,...). Dans ce cas, on a constaté que l'ordinateur est un excellent répétiteur. Or à certains moments, il faut que l'enseignant se change en répétiteur, mais jamais il ne le sera pour chaque élève en particulier.

En faisant abstraction des problèmes de coût qui ne sont jamais éternellement insurmontables, une objection majeure

apparaît toutefois : si l'ordinateur est un excellent outil pédagogique, alors pourquoi est-il le plus souvent laissé au rebut ?
Une des sources du problème proviendrait, à mon avis, d'une ignorance mutuelle : en règle générale, les enseignants ne sont pas formés en vue de l'utilisation pédagogique de l'informatique et les informaticiens, de leur côté, ne perçoivent pas de possibilités de rentabiliser des efforts de développement de didactitiels de qualité.

QU'A-T-ON FAIT DE CET OUTIL ?

Dans l'industrie, l'importance des ressources mises en oeuvre a permis d'obtenir des résultats opérationnels surtout en ce qui concerne l'apprentissage de tâches répétitives mais aussi dans des domaines où les techniques de l'intelligence artificielle sont rentables (expertise, diagnostic,...).
Dans l'enseignement, la situation est différente principalement en raison du manque de moyens. On est presque tenté de dire que l'enseignement général doit 'bricoler' ses didacticiels uniquement grâce à la ténacité et la bonne volonté d'enseignants qui se sont formés à ces techniques de leur propre initiative.
Il est dommage que la possibilité de développer un enseignement assisté par ordinateur digne de ce nom se heurte au refus d'y allouer les ressources nécessaires.
Bien entendu, des firmes privées proposent aussi des didacticiels. Mais dans ce cas, il s'agit bien souvent de constructeurs qui veulent se donner une coloration 'pédagogique' afin d'enlever les marchés d'équipement des écoles en matériel informatique.

Il apparaît donc qu'en raison du manque de didacticiels de qualité, le matériel acquis par les écoles est peu utilisé comme outil et devient "la chose" avec laquelle il faut se familiariser.

Il est peut-être temps de définir ce que l'on entend effectivement par didacticiel. En effet, sous l'appellation fumeuse de "logiciel éducatif", on retrouve à peu près tout ce que l'informatique peut produire : de la calculette sur

ordinateur jusqu'au compilateur pascal en passant par les simulateurs de vol et les éditeurs de texte ! Je crois qu'il faut faire une nette différence entre un compilateur (logo, basic, pascal...) et un didacticiel, logiciel destiné à aider un étudiant à apprendre quelque chose.

En plus des langages classiques (Pascal,...), on retrouve essentiellement deux techniques de développement à l'origine de ces didacticiels, à savoir les langages d'auteurs et des techniques dites d'intelligence artificielle.

Les langages d'auteurs sont en fait des langages de programmation destinés à exprimer les différents états que peut prendre une leçon. Or ce n'est pas un pseudo-langage informatique qui permettra, de par sa seule simplicité, de résoudre le double problème de la maîtrise des aspects informatiques et surtout pédagogiques de la fabrication d'une leçon. Dès que celle-ci devient un peu complexe, sa préparation se mue en une entreprise fastidieuse et longue. Dans la plupart des situations réelles, comme ces langages ne permettent pas de manipuler des 'concepts pédagogiques' de haut niveau, cela revient à demander à l'enseignant de réaliser les Travaux d'Hercule...
Cette approche se révèle donc souvent impraticable.

De ce point de vue, une pratique salutaire serait donc de ne plus confondre le rôle du concepteur de didacticiels et celui de l'utilisateur (l'enseignant dans le cadre d'un cours). Là se pose non seulement le problème de l'adéquation entre l'offre et la demande, mais aussi celui de l'aspect dynamique d'un cours. Il faudrait qu'un enseignant puisse rapidement adapter un didacticiel à l'objet particulier de son cours, ce qui n'est pas vraiment le cas actuellement.

C'est pourquoi certains proposent dès lors de profiter des progrès réalisés en matière d'intelligence artificielle. Ici les problèmes sont d'un autre ordre. Si l'on considère les applications de ces techniques, on constate actuellement qu'elles utilisent souvent des ressources hors de la portée des moyens dont dispose l'enseignement.
D'autre part, l'intelligence artificielle est loin d'être la panacée universelle qui gomme tous les problèmes que l'informatique traditionnelle ne parvient pas à résoudre.

D'aucuns s'ingénient à entretenir la confusion entre les techniques à l'aide desquelles on peut résoudre un problème et la solution à ce même problème : ce n'est pas parce que l'on développe des techniques traitant de l'acquisition de connaissances par la machine que la transposition directe de ces techniques va apporter une solution à la transmission de savoir entre l'enseignant et son élève...

L'enseignement assisté par ordinateur connaît donc des fortunes diverses et se heurte à un certain nombre d'écueils (surtout dans le domaine de l'éducation).
Les acteurs en présence (enseignant, pédagogues, ..., informaticiens) ne savent pas très bien quel est le rôle précis qu'ils ont à jouer les uns par rapport aux autres.
Quand bien même les rôles seraient définis, le texte de la pièce (les concepts pédagogiques) est à définir avec précision parce qu'improvisation et informatisation sont deux disciplines qui cohabitent très mal.
Enfin, sans producteur, il n'y a pas de spectacle, sans la volonté d'y consacrer les ressources nécessaires, l'enseignement assisté par ordinateur restera toujours à l'état embryonnaire....

CAHIER des CHARGES et CONTEXTE

Après avoir vu dans les grandes lignes l'utilisation que l'on a fait de l'ordinateur en tant qu'outil pédagogique, voyons les exigences que devraient rencontrer un didacticiel de qualité.
D'une part, il serait écrit par une personne autre que l'enseignant tout en permettant à ce dernier de le personnaliser pour qu'il cadre avec son cours et ce, moyennant un effort de préparation qui globalement ne soit pas plus important que celui requis pour la préparation d'une leçon non informatisée.
De plus, il devrait être réutilisable pour des leçons présentant certaines similitudes de manière à être rentabilisé au maximum (tant en ce qui concerne le temps consacré à la préparation d'éléments de leçons que du point de vue de l'investissement pécuniaire).

Enfin, le contexte dans lequel on utilise un didacticiel est assez particulier.
Sans prétendre considérer de manière exhaustive tous les aspects d'un logiciel classique et d'un didacticiel, je vais tenter de les comparer pour faire ressortir les principales particularités du contexte de l'enseignement.

Dans une application informatique traditionnelle, on peut distinguer trois acteurs en présence : le concepteur du logiciel, le logiciel lui-même et l'utilisateur au sens large.
Dans ce contexte, on peut dire que l'utilisateur et le logiciel jouent dans un décor (une configuration donnée) pour fournir un service à une organisation.

Dans le cas d'un didacticiel, c'est fort différent, on peut voir ici quatre acteurs en présence : le concepteur du didacticiel, le didacticiel lui-même, l'enseignant et l'étudiant.
Ici, c'est l'enseignant et le didacticiel qui, dans le cadre d'une leçon et à l'aide d'un ordinateur, tentent d'apprendre quelque chose à l'étudiant.
C'est l'importation de la vision de l'informatique traditionnelle qui fait que l'on confond souvent l'enseignant soit avec le concepteur (il faut qu'il puisse modifier le didacticiel pour qu'il entre dans le cadre de sa leçon), soit avec l'utilisateur (didacticiel conçu comme une extension non modifiable du matériel).

Si l'on conçoit le didacticiel comme un outil d'aide que l'on peut dès lors rapprocher des familles de logiciels d'aide à la décision ou de conception assistée par ordinateur, on voit qu'il doit pouvoir être maîtrisé par deux des acteurs et ce, dans des contextes différents : la préparation de la leçon (l'enseignant) et le déroulement de la leçon (l'étudiant).

De plus, je crois qu'il faut éviter de refaire les erreurs commises lors des balbutiements de l'intelligence artificielle. Il est impossible de concevoir un générateur de didacticiels couvrant l'enseignement de n'importe quelle matière, comme il était illusoire de vouloir reconstituer "l'esprit humain" à l'aide d'un ordinateur !

OBJECTIF

Dans le cadre de ce mémoire, j'aborderai ce qui pourrait constituer un élément d'un cours d'électricité assisté par ordinateur : la simulation d'un laboratoire d'électricité. Ceci ne peut pas vraiment être considéré comme un didacticiel complet au sens de la définition que j'ai donnée ci-dessus.
En effet, (si toutefois l'utilisation de l'ordinateur s'avère justifiée) il faudrait y ajouter, pour être complet, d'autres fonctionnalités permettant d'illustrer un principe (loi d'ohm,...), d'en présenter les éléments de théorie ainsi que des exemples d'applications pour enfin aboutir à des exercices pratiques (montages, exercices corrigés,...).

Dans ce cadre, qu'apporte donc l'utilisation d'un ordinateur par rapport aux bonnes vieilles méthodes du tableau noir, de la boîte de résistances, de la fumée et des étincelles des courts-circuits ?
On oublie souvent que le recours à un ordinateur, dans quelque contexte que ce soit, n'est justifiable que par les améliorations qu'il apporte.
Une démarche salutaire consiste à aborder les problèmes en envisageant les différentes possibilités de les résoudre. En présence de plusieurs voies permettant d'aboutir à une solution, il faut alors se pencher sur les moyens dont on dispose, sur les outils au sens large que l'on peut mettre en oeuvre. Comme le plombier devant sa caisse à outils, on opère alors un choix judicieux... qui parfois consiste à se servir d'un ordinateur.
Malheureusement, c'est souvent l'inverse qui se produit, on dispose d'un ordinateur, et l'on cherche quels problèmes on pourrait bien lui faire avaler.

L'objectif de ce mémoire n'est donc pas d'utiliser un ordinateur pour simuler un laboratoire d'électricité !
Mais l'introduction d'un laboratoire d'électricité dans un contexte d'enseignement se heurte à certains problèmes que l'on peut résoudre efficacement grâce à l'ordinateur.

DEMARCHE

Dans un premier temps, je m'efforcerai de situer le problème.

Après avoir dégagé certaines caractéristiques générales de l'utilisation de l'ordinateur dans l'enseignement, je montrerai dans quelle mesure ces caractéristiques rencontrent les problèmes qui surgissent dans un laboratoire d'électricité de l'enseignement secondaire.
Ceci permettra de définir l'objectif à atteindre.

Après avoir défini le problème à résoudre, j'évoquerai brièvement différentes manières de le traiter .
Je dégagerai ensuite la méthode de résolution que j'ai retenue dans le cadre de ce mémoire.

Enfin, il s'agira de traduire cette solution en termes de modules et de structures de données pour finalement discuter les principaux choix d'implémentation qui ont dû être posés.

Deux principes fondamentaux guideront ces deux étapes (solution informatique et implémentation).
La démarche doit être modulaire et son produit doit être réutilisable dans d'autres contextes.

La modularité est ici synonyme d'indépendance maximale entre les différents composants, ce qui leur donne une grande autonomie. Cette aspect doit se doubler d'une grande souplesse d'utilisation qui s'obtient lorsque l'on peut tirer parti de la puissance d'un outil en connaissant uniquement son mode d'utilisation indépendamment de la manière dont il est fabriqué (pour bien utiliser une foreuse, il est inutile d'en connaître les éléments constitutifs).
Cependant, la modularité ne suppose pas automatiquement la réutilisabilité des composants : on peut très bien obtenir un ensemble de composants qui ne peuvent être utilisés que dans le cadre d'UNE seule application tout en respectant les exigences de modularité.
Il m'a semblé que c'était insuffisant, que l'effort de

développement devait pouvoir être récupéré. Pour cela, il faut voir plus loin que les aspects strictement nécessaires à la résolution d'un problème particulier (sans tomber dans l'excès contraire).

Ces exigences m'ont conduit à développer des outils permettant de manipuler un certain nombre d'objets d'usage assez général mais spécifiquement adaptés aux types de problèmes à résoudre.
Chacun de ces composants permet d'apporter un élément de solution au problème particulier qui nous occupe, mais en plus, cet élément de solution présente des caractéristiques assez générales pour aider à résoudre d'autres problèmes.

Cette démarche a été suivie pour tous les composants du didacticiel :

- Communication avec l'utilisateur
  => gestion d'écrans et de menus
- Mémorisation à long terme
  => gestion du stockage de collections d'objets
- Plan d'un circuit électrique
  => gestion de la construction d'un circuit électrique
- Comportement d'un circuit électrique
  => gestion de la simulation d'un circuit électrique
- ....

En ce qui concerne la description des différents modules, je me suis surtout attaché à exposer les idées sous-jacentes à leur développement. J'ai adopté cette démarche à tous les niveaux de conception de manière à offrir une vue d'ensemble (plus ou moins détaillée selon le niveau de développement auquel on se trouve) plutôt que d'exprimer directement dans l'un ou l'autre formalisme classique l'aboutissement de ces mêmes idées.

Une optique similaire fut adoptée pour la présentation des différents aspects de ce mémoire, chaque fois que cela s'avérait possible.

CHAPITRE I

Définition du problème

Ce chapitre tentera tout d'abord de préciser la notion de laboratoire d'électricité et ce, dans le cadre de l'enseignement secondaire.
Ensuite, quelques éléments de théorie seront introduits, de même que les notations qui seront utilisées dans ce travail.
Après avoir défini ce cadre général, la structure du problème sera détaillée de manière à cerner l'objectif à atteindre.
Ceci permettra de dégager la physionomie des types de montages envisagés tant du point de vue des composants utilisés que de leur mode de fonctionnement.
Le problème se trouvera ainsi défini par le type, la taille et les paramètres des montages envisagés.

1.1. Type de laboratoire envisagé

1.1.1. Contexte

Dans l'introduction, j'ai mentionné les liens de parenté que l'on peut trouver entre l'enseignement assisté par ordinateur (EAO) et la conception assistée par ordinateur (CAO).

Relevons tout d'abord quelques aspects généraux de la CAO à travers une de ses applications : le travail d'un ingénieur dans le domaine de l'aéronautique qui consisterait en la conception d'un élément de la carlingue d'un avion. En simplifiant, on peut dire que le résultat du travail de l'ingénieur se traduira par un dessin représentant la pièce et définissant ses caractéristiques. Avant que n'apparaissent des outils de CAO, ce dessin était d'abord agrandi pour être ensuite projeté sur une tôle. Cette dernière était finalement découpée.
On peut actuellement imaginer que les trois dernières étapes que j'ai citées soient entièrement automatisées et qu'un ordinateur commande directement un robot chargé de découper la tôle en suivant les spécifications de la pièce définies par l'ingénieur.

Quand on conçoit un objet, on peut le considérer selon deux angles :
d'une part, ses caractéristiques géométriques
(forme, taille,...)
et d'autre part, ses propriétés
(ce à quoi il est destiné, matière de l'objet, caractéristiques mécaniques,...)

Dans l'application précitée, on retrouve bien ces deux facettes :
- la forme de la pièce de l'élément de carlingue
- ses propriétés (une tôle, élément de carlingue d'avion et non de fer à repasser...)

L'apport d'un outil de CAO doit se situer sur ces deux terrains à la fois par le biais des fonctions qu'il offre.

Il y a d'abord toutes les fonctions de gestion des caracté-

ristiques géométriques des objets parmi lesquelles on retrouve les fonctions de communication graphique (c'est-à-dire tout ce qui peut d'une manière générale aider à écrire ou dessiner sur une feuille, un écran...).
Ensuite viennent toutes les fonctions de modélisation des caractéristiques de l'objet à concevoir parmi lesquelles on retrouve également des fonctions de communication graphique particulières, spécifiquement dédiées au type d'objet à concevoir (formes particulières,...).

Ces caractéristiques modélisées font le lien entre l'objet à concevoir et sa représentation symbolique (symboles graphiques). On peut distinguer deux types de liens, d'une part ceux qui portent sur l'aspect de l'objet (encombrement physique,...) et d'autre part, ceux qui portent sur les propriétés de l'objet (propriétés mécaniques,...).

Ce sont les fonctions de modélisation qui font la spécificité d'un outil de CAO par rapport à un autre.

Enfin, considérons l'itinéraire que doit emprunter la pièce de la carlingue d'avion entre l'imagination de l'ingénieur et la tôle : il se résume à l'élaboration du composant à l'aide d'outils de CAO, comme si l'ingénieur se servait directement d'un chalumeau hyper-sophistiqué.
Ceci met en lumière un troisième aspect de la CAO : le raccourcissement du chemin que doit emprunter l'information : on remplace un niveau d'abstraction (l'épure) par une manipulation plus directe de l'objet à concevoir.
En effet, si on voulait conserver la même démarche, mais sans outil de CAO et en gommant les possibilités d'abstraction du créateur, cela reviendrait à construire un certain nombre de prototypes, les tester, les modifier,...

Le fait de manipuler des objets différents avec intelligence, c'est-à-dire en sachant que tel objet a telles caractéristiques, requiert une approche particulière, adaptée à chaque cas. De plus, pour qu'une application de CAO soit réellement efficace et utile, il faut absolument dépasser le stade de la communication graphique, il faut que l'ordinateur prenne également en charge une partie des aspects de la conception qui sont modélisables.
Le seul moyen d'y arriver, c'est de spécialiser les outils pour qu'ils soient extrêmement bien adaptés à un domaine

restreint d'application.
C'est le même principe que celui qui préside à la différence entre le marteau d'un bricoleur et celui d'un ardoisier. Et rien n'empêche de disposer d'une boîte à outils si cela s'avère nécessaire.

Dans le domaine de l'EAO, on retrouve également les différentes facettes évoquées à propos de la CAO. Prenons un exemple dans le cadre d'un cours de géométrie plane.
Dans ce cas, la facette la plus évidente est le moyen de communication graphique (le dessin des formes géométriques).
Cependant, les autres aspects sont aussi présents. La modéliation d'objets ayant des caractéristiques bien précises (droites, triangles,...) mais aussi la modélisation des notions ayant trait aux relations entre ces objets (parallélisme, perpendicularité, projection,...).
Enfin, l'effet de 'raccourcis' se retrouve par exemple dans la possibilité de construire dynamiquement des démonstrations (l'étudiant élabore la démonstration d'un théorème particulier en manipulant les entités modélisées).

Comme je l'ai dit plus haut, pour obtenir des outils bien adaptés, il faut les spécialiser.
On peut par exemple dégager quatre aspects de l'enseignement d'une matière particulière : la mémorisation (les différents principes utilisés, le vocabulaire,...), l'illustration (montrer l'utilisation que l'on peut faire de ce qui a été mémorisé), l'explication des concepts ('mécanique' sous-jacente aux principes, origines d'un mot,...) et enfin la mise en oeuvre des concepts.
Se limiter à ces quatre aspects pour spécialiser les outils est cependant insuffisant.
De même que l'on a inventé plusieurs types de marteaux en fonction de ce sur quoi on voulait taper, on imagine aisément qu'il faille, par exemple, développer différents types d'outils d'aide à la mémorisation selon le type de matière à mémoriser (vocabulaire / axiomes mathématiques).

Après m'être attaché à montrer les liens de parenté qui existent entre la CAO et l'EAO, il est temps, je crois, de rendre à ce dernier sa spécificité.
Tout d'abord, remarquons que l'enseignant n'est pas un ingénieur et que l'étudiant n'est pas tout à fait une tôle.

La manière d'aborder les problèmes devra donc tenir compte de cette différence. Pour continuer l'analogie, on dira plutôt qu'enseignant et étudiant sont tour à tour dans le rôle de l'ingénieur travaillant la matière, objet de l'enseignement. Comme je l'ai dit dans l'introduction, il ne faut pas non plus confondre l'enseignant avec le concepteur de didacticiel. Il faut donc que chaque application d'EAO puisse être vue de deux manières différentes : par un enseignant et par un étudiant.

1.1.2. Pourquoi simuler un laboratoire d'électricité ?

Pour répondre à cette question, il faut d'abord savoir quelle est la place d'un tel outil de simulation dans le contexte évoqué au point précédent.

On pourrait situer les programmes de simulation en EAO, comme des éléments destinés à apporter une aide à l'enseignant dans l'illustration de principes. De plus, ces programmes de simulation offrent l'avantage de pouvoir également servir d'outils de base pour la correction automatique d'exercices portant sur les mécanismes sous-jacents aux principes qu'ils simulent. Dans ce cas, le didacticiel de simulation pourrait avantageusement être complété par un autre didacticiel qui prendrait en charge le volet explicatif. Enfin, on pourrait se servir des concepts expliqués pour tester la résolution des problèmes.

Maintenant, le but du jeu n'est pas de savoir s'il y a moyen d'utiliser un ordinateur dans ce contexte. Il s'agit plutôt de dégager des éléments susceptibles d'être améliorés et, en tant qu'informaticien, de voir si l'ordinateur constitue une réponse adéquate en tenant compte des contraintes liées à son utilisation.

Pour justifier le développement d'un didacticiel, on peut avancer deux types de raisons. Les premières sont extrêmement classiques et sont toujours avancées lorsqu'il s'agit de justifier l'informatisation de quelque chose. Ce sont tous les arguments liés de près ou de loin à la notion de rentabilité et d'efficacité.
Le second type de raisons que l'on peut invoquer est, quant à lui, lié à la notion de "plus-value" qualitative.

Si l'on se donne la peine de réfléchir, on constate que les arguments de type quantitatif sont tout aussi subjectifs que les arguments de type qualitatif. La seule différence réside dans le fait que les premiers sont sans doute beaucoup plus faciles à manipuler...

Pour ma part, je ne développerai ici que quelques arguments de type qualitatif dont certains peuvent avoir dans un second temps des aspects quantitatifs.

Dans le cadre d'un cours d'électricité, un laboratoire comporte un certain nombre de contraintes qu'il serait utile de voir s'estomper. En voici quelques-unes :

- La construction, les modifications et les manipulations d'un montage peuvent être longues, fastidieuses, répétitives. De plus, au bout d'un certain temps, elles ne présentent plus aucun intérêt dans le cadre du cours.
- A moins de disposer d'un nombre considérable d'appareils de mesure, il n'est possible d'effectuer, sans modification des connexions du montage, qu'un nombre limité de mesures sur une partie seulement des composants du montage.
- En cas de dysfonctionnement d'un montage, il n'est pas toujours évident pour l'étudiant d'en localiser facilement l'origine.
- Les belles étincelles peuvent parfois présenter un certain danger tant pour l'apprenti sorcier que pour le matériel lui-même.
- ...

L'utilisation d'un ordinateur permet justement de rencontrer parfaitement ces objections : il acquiert dans ce cas la qualité d'outil pédagogique pour autant que certains inconvénients ne viennent pas annihiler les bénéfices attendus...
Le principal avantage de simuler un laboratoire d'électricité réside dans le fait que ce dernier devient un élément actif du cours. Il diminue un certain nombre de contraintes qu'il est difficile, voire même impossible de supprimer par d'autres moyens.
On peut également noter que la manipulation de composants modélisés permet parfois de faire plus facilement le lien avec les différents aspects théoriques sous-jacents (au

lieu de manipuler physiquement un circuit, on agit sur la modélisation de ce circuit..).

De plus, relevons un certain nombre d'avantages plus techniques parmi lesquels :

- Au lieu de devoir sans cesse monter et démonter les mêmes circuits, on peut très bien stocker les montages, ce qui les rend réutilisables.

- Grâce à la possibilité d'effectuer des mesures "théoriques", au lieu d'obtenir le module d'une différence de potentiel, on accède à sa représentation complexe comme dans les recueils d'exercices.

- ....

Ces quelques éléments de justification permettent également de cerner les grandes lignes des caractéristiques du laboratoire d'électricité envisagé.
C'est à partir de ces notions que nous allons maintenant préciser l'exposé du problème en posant d'abord quelques définitions sur lesquelles il s'appuie.

1.2. Définitions et éléments de théorie

1.2.1. Notations

Les notations introduites ci-dessous sont relatives aux objets manipulés (résistance, condensateur, self ainsi que les sources).

Une source sera caractérisée par trois éléments : une différence de potentiel (E exprimée en volts V), une intensité (I exprimée en ampères A) et une fréquence (f exprimée en hertz Hz). On distinguera également les sources de tension (E stable) des sources de courant (I stable).
Pour un fonctionnement en mode alternatif, la constante ω est définie comme étant égale au produit $2\pi f$.

Les nombres complexes seront soit sous forme rectangulaire (A+jB) soit sous forme polaire (*module* cis(*angle en degrés*) avec cis(x) = cos(x) + i sin(x)).

La valeur d'une résistance R sera exprimée en ohms Ω, sa représentation comme impédance complexe étant R + j0.

La valeur d'une capacité C sera exprimée en farads F, sa représentation comme impédance complexe étant 0 -j/ωC.

La valeur d'une self inductance L sera exprimée en henrys H, sa représentation comme impédance complexe étant 0 +jωL.

1.2.2. Lois d'Ohm et de Kirckoff

La loi d'OHM

> Il y a proportionnalité entre une différence de potentiel et l'intensité du courant qui la produit (E=RI).

Les deux lois de KIRCKHOFF

> La somme des courants arrivant à un point d'un réseau est égale à la somme des courants partant de ce même point.
> [Loi de Kirckoff relative aux courants]
>
> La somme des tensions appliquées à un circuit fermé est égale à la somme des chutes de tensions dans ce circuit.
> [Loi de Kirckoff relative aux tensions]

Ces trois lois seront à la base des solutions développées plus loin. Elles sont valables pour des tensions continues. On peut montrer qu'elles le sont également lorsqu'on utilise des tensions alternatives. On les généralise de telle sorte qu'au lieu de parler de résistances, on parle en termes d'impédances dans lesquelles circule un courant alternatif.
Les caractéristiques des impédances, des courants et des tensions sont alors définies par des nombres complexes au lieu de nombres réels (cfr. courant continu).

1.2.3. Les 'singularités' d'un montage.

J'entend par 'singularité' d'un montage électrique une des causes pour lesquelles celui-ci ne fonctionne pas exactement de la manière escomptée.

Dans le cadre de la simulation, ces 'singularités' peuvent être de plusieurs ordres :
- physique
- modélisation de la réalité physique
- informatisation de cette modélisation.

Les 'singularités' d'ordre physique sont par exemple les courts-circuits, l'utilisation de composants de valeur inadéquate,...
En ce qui concerne la modélisation de la réalité physique du circuit, elles sont souvent l'expression d'une 'singularité' d'ordre physique mais il peut également s'agir d'une incompatibilité entre le modèle mathématique des composants (et de leurs interactions) et leur comportement réel.
Les 'singularités' d'ordre informatique sont, quant à elles, liées à une incompatibilité entre le type de montage que l'on décrit et les capacités limitées de la machine (espace mémoire, précision numérique, puissance des algorithmes mis en oeuvre).
Par extension, on y rangera également les défaillances possibles du tandem ordinateur-didacticiel.

Par la suite, chaque fois qu'il sera fait référence aux 'singularités' d'un montage ou d'un circuit, il faudra donc l'entendre au sens large défini ci-dessus.

1.3. Objectif à atteindre

Dans le contexte décrit au point un, le but est de simuler un laboratoire d'électricité capable de modéliser des montages dont la complexité est semblable à celle des exercices proposés dans des recueils classiques (série Schaum,...).
L'objectif consite en la simulation du comportement d'un montage qui serait effectué à l'aide de fils, boîte à

résistances... Ce ne sera pas un didacticiel directement destiné à superviser des exercices corrigés.
Il doit plutôt avoir les fonctionnalités d'un amas de fils que celles d'un beau dessin sur une feuille de papier.
Il ne s'agit pas en effet d'un langage statique permettant de "disserter" au sujet de circuits électriques, mais plutôt d'une représentation d'éléments avec lesquels on peut jouer à sa guise pour qu'ils finissent par constituer l'un ou l'autre circuit.

La justification du développement d'un tel didacticiel réside d'une part, dans les avantages que j'ai cités plus haut et d'autre part, dans le fait que la simulation du montage doit permettre de localiser les anomalies par lesquelles il peut être affecté.
En effet, dans la mesure du possible, le didacticiel devra être à même de signaler l'endroit où se situe probablement une 'singularité' du montage (cfr 1.2.3).

Bien entendu, il doit aussi offrir la possibilité d'être réutilisé comme composant d'une boîte à outils permettant de développer d'autres didacticiels utilisables dans le cadre d'un cours d'électricité.
Cette possibilité doit aussi exister pour les différents éléments qui le composent.
Il devra donc être constitué d'un ensemble de modules possédant des caractéristiques assez générales pour pouvoir être réutilisés dans un contexte voisin, tout en veillant à être assez spécifiques pour ne pas trop perdre en efficacité. C'est ce dernier aspect qui dictera aussi le type de montage qu'il sera possible de simuler. En effet, au plus le nombre de possibilités sera grand, au plus le didacticiel deviendra complexe (nombre de composants et modes de fonctionnement du circuit). Or cette complexité va à l'encontre de la spécialisation des outils EAO qui doit leur conférer souplesse et efficacité.

1.4. Type de montage

Etant donné le contexte et l'objectif à atteindre, il devient possible d'envisager le type de montage dont on va simuler le comportement.

Dans un laboratoire d'électricité, on peut caractériser un montage par sa complexité (nombre de composants qui le constituent), par son mode de fonctionnenemt ainsi que par le type de composants et par le type de sources d'alimentation.

Dans ce cas-ci, nous nous limiterons à des composants passifs (RLC) pour deux raisons.
La première est que la simulation d'un réseau électrique comportant des éléments actifs (diodes, transistors) passe souvent par sa linéarisation autour d'un point de référence. Ceci va à l'encontre de l'objectif de localisation des 'singularités' du montage.
La deuxième raison est que l'on veut se doter d'un outil spécialisé et qu'une approche globale permettant la simulation de composants actifs et passifs est en contradiction avec la contrainte de spécialisation (les méthodes utilisées pour aborder les deux problèmes sont fort différentes).
De plus, comme nous sommes dans le cadre de l'enseignement, le nombre de composants d'un montage ne doit pas être trop important. Le but est de simuler un montage simple pouvant servir d'illustration à un principe, il ne s'agit pas d'un laboratoire dans le cadre de la conception de montage assitée par ordinateur. Le but est uniquement pédagogique.
Même si plus haut j'ai relevé quelques traits communs entre l'EAO et la CAO, il ne faut pas perdre de vue que les finalités sont différentes.

Les sources d'alimentation du montage seront des sources de tension ou des sources de courant travaillant à une fréquence déterminée. Le montage utilisant une source continue ne pourra cependant comporter que des résistances.

La simulation du comportement du montage sera limitée à l'état stable. Pour traiter les transitoires, il faudrait utiliser une toute autre approche que celle qui est emprun-

tée dans ce travail. Ceci va tout à fait dans le sens de la spécialisation des outils que j'ai prônée plus haut.

1.5. Physionomie du problème

De ce qui précède, on peut dégager les caractéristiques générales du problème à résoudre.

Tout d'abord, voyons avec précision ce que l'on entend par "comportement d'un montage électrique".
Dans le cadre de ce mémoire, étant donné que l'on se limite à l'état stable, l'observation du comportement d'un montage se réduit à mesurer le courant qui circule dans les composants ainsi que la différence de potentiel générée par ce courant.

De manière générale, il s'agit donc de simuler le comportement d'un montage constitué de composants passifs (résistances, condensateurs, selfs) et alimenté par une ou plusieurs sources (de tension ou de courant) alternatives ou continues.

De plus, les moyens qui auront permis de simuler le comportement global du montage devront être accessibles d'une part, à l'utilisateur (manipulation de concepts) et d'autre part, comme éléments d'une boîte à outils.

Cela donne en définitive une articulation du problème à trois niveaux.
Le premier est celui de la construction du montage, le second est constitué par la simulation du comportement de ce montage, le troisième réside dans la possibilité de réutiliser les différents mécanismes et fonctionnalités des deux niveaux précédents.

CHAPITRE II

Stratégie de résolution

Le problème étant défini, ce chapitre aura pour but de détailler la stratégie de résolution adoptée.
Tout d'abord, des méthodes classiques de résolution seront brièvement présentées. Ensuite, on abordera la méthode retenue dans le cadre de ce travail.
Celle-ci se présente sous deux aspects, l'un plutôt topologique et l'autre plutôt numérique.
Grâce à la combinaison de ces deux aspects, il devient possible de résoudre le type de problème défini plus haut tout en permettant dans une certaine mesure, de localiser les 'singularités' du circuit.

2.1. Méthodes classiques de résolution

2.1.1. Application des deux lois de Kirckoff

Ces méthodes se basent soit sur la loi de Kirckoff relative aux courants (méthode des Noeuds) soit sur la loi de Kirckoff relative aux tensions (méthode des Mailles). Toutes deux aboutissent à la constitution de la matrice d'un système d'équations que l'on résoud ensuite par une méthode directe (Gauss,...) ou par une méthode itérative (Gauss-Seidel,...).

2.1.2. Kirckoff et la théorie des graphes

Dans les méthodes précédentes, on exploite très peu la structure topologique du circuit. En se basant sur la théorie des graphes, on fait ressortir les aspects topologiques contenus dans les deux lois de Kirckoff. Si le circuit est considéré comme un graphe, il peut être vu comme un ensemble de sommets (cfr méthode des Noeuds) mais aussi comme un ensemble d'arcs reliant ces sommets (cfr méthode des Mailles). Bien entendu, il faut aussi tenir compte de la nature des composants, ce qui fait l'objet d'un troisième angle de vue.
En analysant le circuit dans ces trois directions, on obtient de nouveau un système d'équations regroupant deux types d'équations qui correspondent d'une part, au circuit vu comme un ensemble de sommets reliés par un ensemble d'arcs et d'autre part, au circuit vu comme un ensemble de composants.

2.1.3. Que retenir de ces méthodes ?

Dans le cadre du problème qui nous occupe, les méthodes citées plus haut mènent toutes à la résolution d'un système d'équations basé sur l'observation du circuit à étudier.
Elles permettent d'avoir une approche du problème dans toute sa généralité.
Cependant, du point de vue de l'étudiant, il est très difficile de faire le lien entre un système d'équations et la signification physique des principes sur lesquels il se

base. Or, dans une optique pédagogique, il me paraît indispensable d'avoir une approche qui soit directement compréhensible par l'étudiant dans un but d'accompagnement de la démarche de résolution du problème.
De plus, de telles méthodes ne permettent pas de résoudre le problème de la localisation des singularités.
C'est pourtant vers de telles méthodes qu'il faudrait se tourner si l'on voulait aborder des cas d'application plus larges comme le problème des transitoires.

2.2. Etude d'un circuit

2.2.1. Principes théoriques

Imaginons que nous ayons à étudier un circuit électrique quelconque formé d'un certain nombre d'éléments (sources, résistances, condensateurs, selfs).
Pour mener à bien cette étude, nous disposons d'une part, de quoi dessiner des circuits autant de fois que cela s'avérera nécessaire (un bloc de feuilles de papier) et d'autre part, de moyens de calculs efficaces mais d'une puissance limitée (une bonne calculette).

Le premier obstacle à franchir est constitué par la multiplicité des sources d'alimentation. Qu'à cela ne tienne, le principe de superposition nous dit qu'il suffit de résoudre le problème pour chaque source séparément, en remplaçant les autres soit par un circuit ouvert (source de courant) soit par un court-circuit (source de tension).
Il ne reste plus ensuite qu'à superposer les différentes solutions obtenues pour chaque source (somme des différentes solutions).
Nous nous baserons donc sur ce principe dès que nous rencontrerons plusieurs sources, ou bien dès qu'une source ne générera plus une tension sinusoïdale. Cette dernière sera remplacée par l'approximation de sa forme au moyen d'un des premiers termes du développement en série de Fourier de plusieurs tensions sinusoïdales.
Nous voilà donc revenus à une situation où un circuit est alimenté par une source sinusoïdale unique.

Une question se pose alors : existe-t-il une possibilité de réduire la complexité du circuit pour en arriver à la situation élémentaire d'un unique composant alimenté par une source.
Une réponse positive à cette question consiste à trouver un moyen de remplacer le circuit par un composant équivalent connecté aux bornes de la source d'alimentation (on s'occupe du comportement d'un élément du circuit vu uniquement de l'extérieur).
Pour y arriver, il faut faire "disparaître" deux types d'objets : les composants (branches) et les points de jonction (noeuds).
Dans un premier temps, on dispose de deux moyens simples : remplacer par leur équivalent, les composants qui sont en parallèle ou en série.
Malheureusement, cela ne suffit pas, on ne pourra pas toujours faire disparaître de la sorte un noeud auquel aboutissent plus de deux branches.
A. Rosen (cité par Benson & Harrison) a montré que toute étoile constituée de *n* branches peut être remplacée par *n(n-1)/2* branches reliant deux à deux les extrémités des branches de l'étoile. (généralisation de la transformation étoile-triangle où dans ce cas *n*=3).

En appliquant ce principe, il est donc toujours possible de supprimer un noeud d'un circuit. Si après chaque réduction d'étoile, on regroupe les composants qui sont en série et en parallèle, on aboutira après un certain temps au montage élémentaire recherché.

Ayant un composant unique relié à une source d'alimentation, il devient donc trivial de connaître son comportement au sens où on l'a défini plus haut. On déduira donc le courant traversant ce composant et ainsi la différence de potentiel qui existe entre ses bornes (ou inversement).

Cependant, il faut maintenant être certain de pouvoir répercuter ces informations vers tous les composants "cachés à l'intérieur" du composant équivalent.

Pour cela, il suffit de voir si toutes les transformations qui nous ont conduits à ce composant équivalent sont réversibles.

En ce qui concerne les deux premières, on appliquera les deux lois de Kirckoff (série-loi relative aux courants, parallèle-loi relative aux tensions) combinées avec la loi d'Ohm (connaissant le courant traversant un composant, on en déduit la différence de potentiel qui règne à ses bornes et inversement).

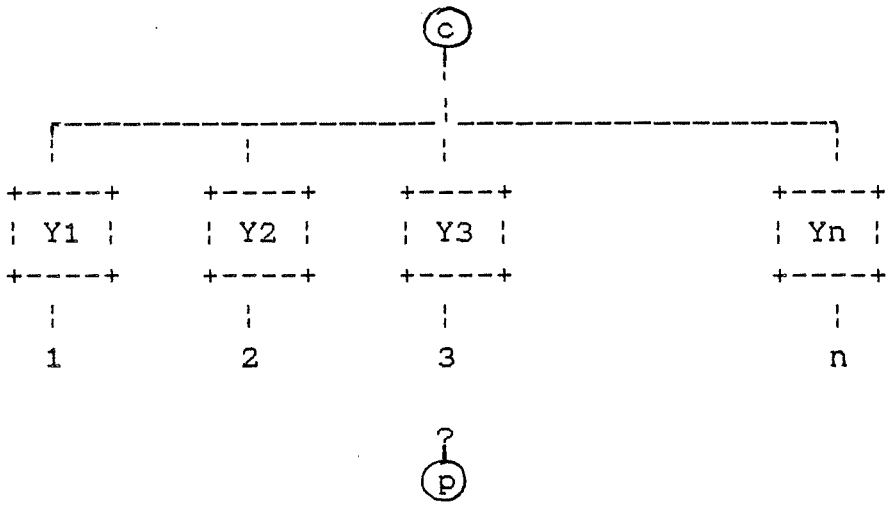


Figure 2.1

Pour ce qui est de la transformation étoile-triangle généralisée, ce n'est pas aussi évident.
L'application du théorème de Millman (cité par Benson & Harrison) nous tranquillisera de manière définitive. Selon lui, connaissant la valeur de *n* admittances (inverse de l'impédance : cfr Yi de la figure 2.1) connectées à un noeud commun (c), il n'est pas nécessaire de connaître la manière dont leur autre extrémité (i=1..n) est reliée à un point quelconque du circuit (p) pour déterminer la différence de potentiel qui existe entre le noeud commun (c) et le point considéré (p) : il suffit de connaître la différence de potentiel qui règne entre chaque extrémité (i=1..n) et le point quelconque considéré (p).
Plus loin, nous verrons en détails comment appliquer ce théorème en choisissant le centre de l'étoile et l'extrémité d'une de ses branches respectivement comme le point c et le point p.

De tout ce qui précède, nous pouvons conclure qu'il est possible de réduire un circuit alimenté par une source

unique à un composant élémentaire équivalent. Ceci permet de déterminer le courant qui le traverse et la différence de potentiel qui règne à ses bornes pour ensuite propager ce résultat en l'éclatant vers tous les composants du circuit.

2.2.2. Application

En appliquant les principes généraux évoqués ci-dessus, il est donc possible de simuler le comportement de chaque élément constitutif d'un montage.

Remarquons, que ce type d'approche présente deux facettes : *topologique* et *numérique*.

La première consiste en un parcours du circuit dont le but est de tirer un maximum d'informations à partir de sa structure. Ces informations devront servir à déterminer par exemple quel type de transformation appliquer et à quel endroit,...
La seconde consiste en la mise en oeuvre des formules mathématiques dérivées des principes utilisés. Tout en fournissant un modèle du comportement des composants touchés par la transformation, elles apportent également des informations utiles à la méthode de résolution : choisir parmi plusieurs transformations de même nature, détecter des instabilités,...

Dans le point précédent, nous avons parlé de la généralisation de la transformation étoile-triangle. Dans la suite de cet exposé, nous nous attarderons uniquement à la discussion du cas où les étoiles ne comportent pas plus de trois branches.
Par souci de clarté, il m'a semblé opportun de me limiter à ce cas d'application plus concret.
Cela ne nuit en rien à la portée générale de l'exposé, puisque l'on a montré que les principes sous-jacents permettent d'étudier des cas plus complexes.

2.3. Approche topologique

Les informations que l'on peut tirer de l'observation d'un circuit sont de deux ordres : "électriques et géographiques".
Cependant, elles se déduisent uniquement de la configuration du circuit sans présumer de la nature des composants (résistance, condensateur ou self), en tenant compte uniquement de l'orientation du courant généré par la source.
On considérera donc qu'un circuit est constitué d'un ensemble de noeuds reliés entre eux par des branches, les composants.

2.3.1. Parcours du graphe du circuit

Le parcours du graphe du circuit a pour but de baliser ce dernier avec une double finalité.

D'une part, il s'agit de définir pour chaque composant le sens conventionnel du courant qui le traverse. Cela se fait de manière totalement arbitraire. Néanmoins, on peut aussi essayer de le définir de manière à imiter la démarche de quelqu'un se trouvant devant un schéma électrique (partir de la source d'alimentation en propageant le sens conventionnel choisi pour cette dernière vers les autres composants).

D'autre part, il est utile de définir une estimation de la distance à laquelle se trouve un composant par rapport à la source : ce sera le critère de base de la progression dans le graphe pour l'étude du circuit (quand on veut simplifier le schéma d'un circuit électrique, il est parfois préférable de commencer par les éléments les plus éloignés de la source : comme aucune règle précise n'existe en ce domaine, on peut choisir celle-là puisque dans certains cas, elle peut être plus avantageuse qu'une autre).

Un moyen simple de rencontrer les deux points précédents est d'utiliser une méthode d'exploration en *largeur d'abord* : il suffit pour cela de prendre la source d'alimentation comme point de départ.

Lors de cette exploration, on procédera à la découpe du circuit en étages et ce, en deux temps : d'abord en partant dans le sens du courant généré par la source (sens direct), ensuite dans le sens inverse.

Un étage se définit de manière itérative comme une collection de noeuds reliés par au moins une branche à un même étage précédent, le premier étage se réduisant aux noeuds auxquels sont directement reliées les bornes de la source. Chaque étage est donc divisé en deux classes, la première correspondant aux noeuds atteints dans le sens direct, la seconde correspondant aux noeuds atteints dans le sens inverse (à un même étage, en cas de conflit lors de la détermination du sens du courant, le sens direct l'emportera toujours, par convention, sur le sens inverse).

2.3.2. Reconnaissance d'éléments de circuit

Le but étant de réduire le circuit, il est donc essentiel de pouvoir construire des agrégats sur base de différentes transformations.
Pour cela, il faut disposer de critères pour les reconnaître. Ceux-ci sont tous basés sur l'observation des jonctions entre éléments du circuit.

Deux branches sont parallèles si elles partagent la même origine et la même destination, elles seront en série si elles sont les seules à aboutir à un même noeud.
De plus, si à un noeud n'aboutit qu'une seule branche, on peut définitivement éliminer cette dernière du circuit (et donc le noeud qui est alors totalement détaché du circuit), il n'exerce en effet aucune influence sur le comportement du circuit.
Enfin, une branche ayant ses deux bornes connectées à un même noeud indiquera la présence d'un court-circuit éventuel.
Comme on l'a vu dans le point deux, la réduction du circuit se base également sur la transformation étoile-triangle. Après avoir épuisé toutes les possibilités d'agrégation parallèle ou d'agrégation en série, une étoile sera facilement reconnue à partir de son noeud central d'où partent exactement trois branches.

Ces observations donnent aussi un moyen simple de contrôle si on limite le nombre de branches qui peuvent constituer une étoile (posons cette limite à *n*).
En effet, si dans un circuit, après avoir agrégé tous les composants qui pouvaient l'être, on ne trouve que des noeuds auxquels sont reliées plus de *n* branches, le circuit ne peut être étudié que grâce à l'utilisation de transformations plus complexes permettant de réduire des étoiles comportant plus de *n* branches.

2.3.3. Sens du courant

La détermination du sens du courant n'a aucune importance pourvu que la convention prise pour un élément du circuit soit respectée tout au long de l'étude du circuit considéré.
Comme ce choix ne repose pas sur des critères absolus, il faudra certainement permettre qu'au moment de la lecture des résultats, l'utilisateur puisse opérer un choix différent de celui qui aura été posé automatiquement.
Il faut néanmoins rester vigilant pour ne pas aboutir à des incohérences lors de la superposition de plusieurs solutions : dans ce cas, le sens du courant ne pourra être déterminé qu'une seule fois tout au long de l'étude du circuit.

2.4. Approche numérique

2.4.1. Nombres complexes

Comme nous allons travailler avec les représentations complexes des impédances, des tensions et des courants, il est utile d'en voir les implications.

Etant donné que les formules utilisées ci-dessous sont assez simples, nous ne nous intéresserons qu'aux quatre opérations de base.

Remarquons tout d'abord que la représentation joue un rôle non négligeable dans la complexité de la formulation des

quatre opérations. La représentation polaire convient parfaitement pour les multiplications et les divisions, tandis que la représentation rectangulaire convient mieux pour les additions et les soustractions.
Habituellement, on utilise la représentation adéquate en fonction des opérations à appliquer.
D'un point de vue logique, c'est peut-être défendable, mais dans la pratique, cela l'est beaucoup moins.
En effet, la conversion d'une représentation à l'autre est d'un coût relativement élevé puisqu'elle nécessite l'utilisation de fonctions trigonométriques.
C'est pourquoi il me semble raisonnable de choisir une seule représentation et de la garder tout au long des calculs, réservant les éventuelles transformations pour deux moments précis : l'acquisition des données et la présentation des résultats.

Dans le cadre de cet exposé, j'ai choisi la représentation qui exige l'emploi des opérations les plus simples sur les composantes réelles et imaginaires des nombres complexes, à savoir la représentation rectangulaire.

2.4.2. Transformations

Nous allons maintenant nous pencher sur la mise en oeuvre des formules mathématiques liées aux transformations que nous utilisons. Rappelons que ces transformations doivent être réversibles. C'est donc également de ce point de vue que nous les étudierons. Souvenons-nous enfin que les éléments que nous manipulons sont des impédances complexes.

La première et la plus simple est la transformation de deux éléments reliés en série (S1,S2) en un élément équivalent (S).
La réduction consiste en une simple addition des valeurs des impédances(S=S1+S2).
La propagation du résultat obtenu pour S (E,I) se base sur la loi de Kirckoff relative aux courants, ce qui donne (I1=I2=I). Reste à trouver la différence de potentiel qui règne aux bornes de S1 et S2 (E1,E2).
Dans un premier temps, on applique la loi d'Ohm (E1=S1*I1).
Ensuite, on a le choix entre une seconde application de la même loi et, plus simplement, l'application de la loi de

Kirckoff relative aux tensions, ce qui donne E2=E-E1.

Le second type de transformation consiste à trouver l'équivalent (P) de deux éléments reliés en parallèle (P1,P2).
Pour les réduire, il suffit d'appliquer la formule suivante : P=(P1*P2)/(P1+P2).
La propagation du résultat obtenu pour P (E,I) se base cette fois sur la loi de Kirckoff relative aux tensions, ce qui donne (E1=E2=E). Reste à trouver le courant qui traverse P1 et P2 (I1,I2).
Dans un premier temps, on applique la loi d'Ohm (I1=E1/P1).
Ensuite, on a de nouveau le choix entre une seconde application de la même loi et, plus simplement, l'application de la loi de Kirckoff relative aux courants, ce qui donne I2=I-I1.

Les deux transformations précédentes sont facilement généralisables à plus de deux éléments, soit en généralisant les formules exposées ci-dessus, soit en appliquant de manière itérative les formules valables pour deux éléments.

Passons maintenant à la transformation étoile-triangle.
Celle-ci consiste à supprimer un noeud du circuit (le centre de l'étoile) en remplaçant les n branches qui en sont issues par n*(n-1) branches reliant deux à deux les extrémités des branches de l'étoile.
Pour une étoile à trois branches, il n'y a pas d'augmentation du nombre de branches, cependant pour des étoiles plus complexes, ce nombre a la fâcheuse particularité d'augmenter assez rapidement (3->3, 4->6, 5->10,...).
Nous aborderons donc le problème de la transformation étoile-triangle sans tenir compte du fait qu'il n'y a pas dans ce cas particulier, d'augmentation du nombre de branches et ce, afin que la démarche proposée garde une portée plus universelle.

En nous rapportant à la figure 2.2 (a), il s'agit de remplacer les branches (*-i) pour i=(1,2,3) par les branches (1-2), (1-3), (2-3).

La formule de transformation s'articule en deux parties, dont une est commune aux trois nouvelles branches.

1) En posant Zi pour i=(1,2,3) comme étant l'impédance de la branche (*-i), on pose D=(Z1*Z2)+(Z1*Z3)+(Z2*Z3)

2) Ce qui donne finalement pour chaque nouvelle branche :

$$Z12=D/Z3$$
$$Z13=D/Z2$$
$$Z23=D/Z1$$

Avec Zij, l'impédance de la branche reliant le noeud i au noeud j.

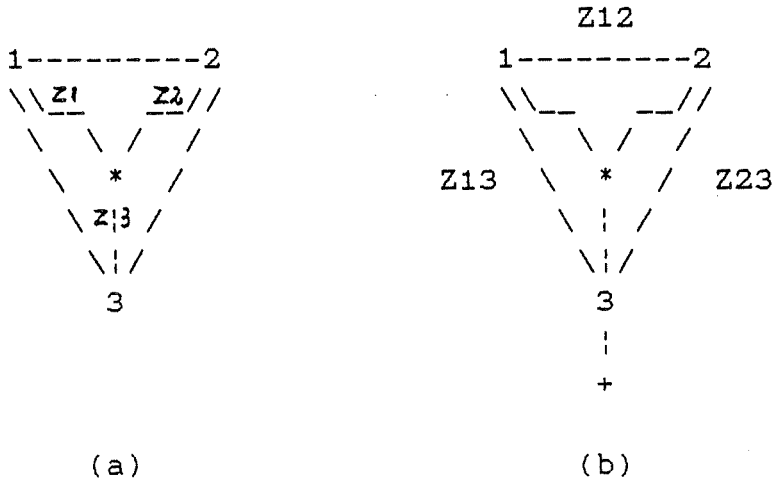


Figure 2.2

Le chemin étant parcouru dans un sens, voyons maintenant comment répercuter les résultats obtenus pour Z12, Z13 et Z23 sur les branches Z1, Z2 et Z3 (cfr fig. 2.2 (b)).
Nous allons appliquer le théorème de Millman en choisissant comme noeud d'où partent n admittances, le centre de l'étoile. Le but recherché est de déterminer la différence de potentiel et le courant circulant dans chacune de ces branches alors que nous connaissons le comportement des branches équivalentes.
Lors de l'application du théorème de Millman, il faut choisir un second point du circuit (+) pour lequel on connaît la différence de potentiel qui règne entre lui et chaque extrémité des branches de l'étoile.
Pour ce faire, il suffit de le relier directement à l'une de ces extrémités (ici 3). La différence de potentiel qui règne entre (+) et (3) est évidemment nulle.

Remarquons que la valeur des admittances reliées au centre de l'étoile (Y1, Y2, Y3) s'obtient facilement comme l'inverse des impédances (Y1=1/Z1, Y2=1/Z2, Y3=1/Z3).
Nous disposons donc de tous les éléments nécessaires à l'application de la formule dérivée du théorème de Millman.

$$V(*+)=[(V13*Y1)+(V23*Y2)+(0*Y3)]/(Y1+Y2+Y3)$$

Ce qui donne en utilisant les valeurs des impédances :

$$V(*+) = \frac{[(V13/Z1)+(V23/Z2)+(0/Z3)]}{[((Z2*Z3)+(Z1*Z3)+(Z1*Z2))/(Z1*Z2*Z3)]}$$

En simplifiant et en reprenant l'expression de D définie plus haut pour la transformation, on obtient le résultat suivant :

$$I3 = \frac{V(*+)}{Z3} = I(*+) = \frac{[(V13*Z2)+(V23*Z1)]}{D}$$

Les courants I1 et I2 sont obtenus de la même manière, les différences de potentiel (*+) se déduisant par une simple application de la loi d'Ohm.

L'approche développée ci-dessus est tout à fait générale, elle peut donc aussi être adoptée pour résoudre des cas plus complexes d'étoiles à plus de trois branches.
Cependant, dans ce cas précis, si on tient compte de la structure triangulaire entourant l'étoile, l'observation du sens du courant et l'application des lois de Kirckoff rendent possible, par simple sommation, la déduction des courants circulant dans les branches issues du centre de l'étoile.
Il en va de même pour les différences de potentiel.

2.4.3. Problèmes des erreurs d'arrondis

Les erreurs d'arrondis sont indissociables de toute représentation discrète d'une grandeur continue : en l'occurence ici, des nombres réels représentés en mode virgule flottante.

Le moyen le plus simple et le plus efficace d'éviter qu'elles ne s'amplifient trop est de réduire tant que faire se peut le nombre des calculs !
Evidemment, il y a toujours un moment où l'on ne peut éviter d'exécuter l'un ou l'autre calcul. Dans ce cas, il reste à contrôler le processus pour voir s'il ne déraille pas.
Connaissant la précision de la représentation des nombres qui est adoptée, et donc l'incertitude avec laquelle la machine "connaît" ce nombre, on peut suivre pas à pas toutes les transformations que l'on fait subir à ce nombre, en propageant l'incertitude de départ selon certaines règles.
Ceci est assez lourd, le mieux est de pouvoir estimer globalement le facteur amplificateur de chaque transformation sur l'incertitude de départ et de comparer cette estimation avec la valeur courante du résultat de la transformation.
Si ces quantités sont d'un même ordre de grandeur, il y a de fortes chances que le résultat que l'on vient d'obtenir ne signifie plus grand chose.

Une deuxième mesure de détection consiste à voir si les nombres que l'on manipule ne sont pas trop proches de l'ordre de grandeur de l'imprécision liée à leur représentation.

2.5. Combinaison des deux approches

2.5.1. Critères de décision

Lors de la réduction du circuit, il faut à tout moment choisir la transformation que l'on va appliquer. Ce choix est double, d'une part déterminer le type de transformation à effectuer et d'autre part, si plusieurs s'avèrent possibles, opérer une sélection.

Le premier critère que l'on va se donner repose sur le principe suivant : appliquer au circuit la transformation la plus simple possible. Comme on manipule des impédances, en partant de la plus simple, on obtient l'ordre suivant :

agrégation série, agrégation parallèle, transformation étoile-triangle (et la simplification des étoiles comportant plus de trois branches).

Le second critère tient compte de l'éloignement de la source, si l'on a le choix entre plusieurs transformations d'un même niveau de complexité, on préférera la transformation affectant les composants de l'étage le plus éloigné de la source. Cette option est un peu arbitraire et guidée simplement par la constatation suivante : la modification des étages les plus éloignés de la source permet parfois de simplifier les transformations ultérieures des étages plus proches. L'application aveugle de transformations complique parfois le circuit, ce qui a dès lors pour conséquence d'empêcher l'application de certaines transformations plus simples.

Le troisième critère permet de départager les candidats d'un même étage sur base de considération d'ordre numérique : la transformation la meilleure du point de vue de la stabilité numérique.
Remarquons ici que l'utilisation du second critère diminue considérablement les occasions d'utiliser le troisième. Une variante intéressante serait la combinaison des deux critères qui tiendrait compte à la fois de l'éloignement de la source et de la stabilité numérique.

Le choix de l'application de l'un ou l'autre critère définit une politique d'étude du circuit. Par exemple, si le but est de savoir le plus vite possible si un circuit est mauvais, on pourrait prendre à rebours le troisième critère de manière à aboutir plus rapidement à une impasse si cette dernière existe.

2.5.2. Description générale de la stratégie.

Ayant un circuit à analyser, on se concentre sur chaque source séparément en supprimant l'influence des autres (cfr principe de superposition).
Disposant d'un circuit alimenté par une source unique, on le découpe en 'étages' (cfr 2.3.1) et ensuite on lui applique une série de transformations visant à obtenir son équivalent élémentaire.

A un moment donné, le type de transformation sera choisi en fonction des différents critères énoncés ci-dessus (transformation la plus simple [1], la "meilleure" [2 & 3]).

Une fois le circuit réduit à un composant unique, on propage le résultat obtenu de manière à déterminer pour chaque composant le courant qui le traverse et la différence de potentiel ainsi générée.

Vient ensuite l'étape de superposition si le montage comporte plus d'une source de courant.

Remarquons toutefois que si le but était simplement de connaître la valeur du composant équivalent, l'étape de propagation peut être supprimée.

2.5.3. Evaluation de la stratégie

La stratégie de résolution proposée ci-dessus permet d'étudier des montages électriques relativement complexes. De plus, comme elle se passe en deux phases, il est souhaitable que toutes les étapes soient mémorisées. Ceci permet de retrouver dans le plan initial du montage, l'endroit où se situe chaque transformation.

L'exigence de localisation des 'singularités' du montage est donc ainsi directement rencontrée.
Il existe cependant des lacunes. Celles-ci sont dues principalement au remplacement d'ensembles de composants par des composants équivalents. Et, plus une 'singularité' sera détectée tardivement, moins sa localisation dans le montage initial sera précise.
Si la 'singularité' est détectée pendant la dernière étape (composant élémentaire), il est bien évident que le fait de signaler qu'il y a un problème quelque part dans le circuit n'est pas d'une grande utilité. Mais dans les autres cas, même si l'indication n'est pas très précise, elle constitue néanmoins un élément utile à la recherche de la cause de l'erreur.

En ce qui concerne l'efficacité de la stratégie, des problèmes ne peuvent surgir que lorsque le montage est d'une complexité trop grande. Ils peuvent être de deux natures.
Il y a d'une part, les problèmes liés aux instabilités numériques qui pourraient se produire lorsque la chaîne des calculs nécessaires à la résolution du circuit est trop longue (propagation des erreurs d'arrondis). Mais ces instabilités peuvent également avoir pour origine une 'singularité' physique : le fait de pouvoir les localiser constitue un grand avantage.
D'autre part, on peut rencontrer des circuits qui, à cause de leur complexité trop grande, ne trouveraient pas d'équivalent unique si on utilise simplement les transformations étoile-triangle du niveau de généralité que l'on a prévu pour le simulateur. En effet, si l'on a décidé la simplification d'étoiles comportant au plus n branches, il devient impossible d'étudier un circuit qui, à un instant donné au cours de la réduction, ne comporte que des étoiles constituées de plus de n branches.

On peut étudier la convergence de la stratégie de résolution proposée en prenant comme critère de convergence vers la solution (pendant la phase de réduction), le nombre total d'éléments que comporte le circuit étudié à un moment donné (nombre de branches + nombre de noeuds).
Dans ce cas, les transformations modifient le nombre d'éléments comme suit :

- transformation série
  (2 branches et 3 noeuds) -› (1 branche et 2 noeuds)
- transformation parallèle
  (2 branches et 2 noeuds) -› (1 branche et 2 noeuds)
- transformation étoile-triangle
  (3 branches et 4 noeuds) -› (3 branches et 3 noeuds).

Si l'on réduit des étoiles de plus de trois branches, ce type de transformation ne permet plus d'obtenir une convergence continue selon le critère défini ci-dessus puisque le nombre d'éléments du circuit n'est plus décroissant.

CHAPITRE III

Modules et Données

Dans les points précédents, la définition du problème a permis de trouver une manière de le résoudre. Ce chapitre-ci envisagera la traduction de la solution retenue sous forme d'algorithmes.
Dans un premier temps, une présentation générale du didacticiel permettra de dégager ses principaux constituants.
Ensuite, en partant de ce que voit l'utilisateur, on détaillera les structures de données sous-jacentes aux objets manipulés.
L'utilisation de ces objets dans le but de simuler le comportement d'un circuit électrique nous amènera dès lors, à dégager la physionomie des principaux algorithmes sur lesquels se base le didacticiel.

3.1. Physionomie de l'outil

Le didacticiel se présente sous la forme d'un ensemble d'outils intégrés directement utilisables. Si on considère globalement cet ensemble, on peut le diviser en deux parties, l'une chargée de la construction du montage, l'autre chargée de modéliser le comportement de ce dernier. En plus de ces deux fonctionnalités, il faut évidemment pouvoir communiquer avec l'utilisateur (clavier, écran,...) et conserver certaines données (fichier).
Ceci nous donne donc cinq modules principaux (cfr rectangles de la figure 3.1)

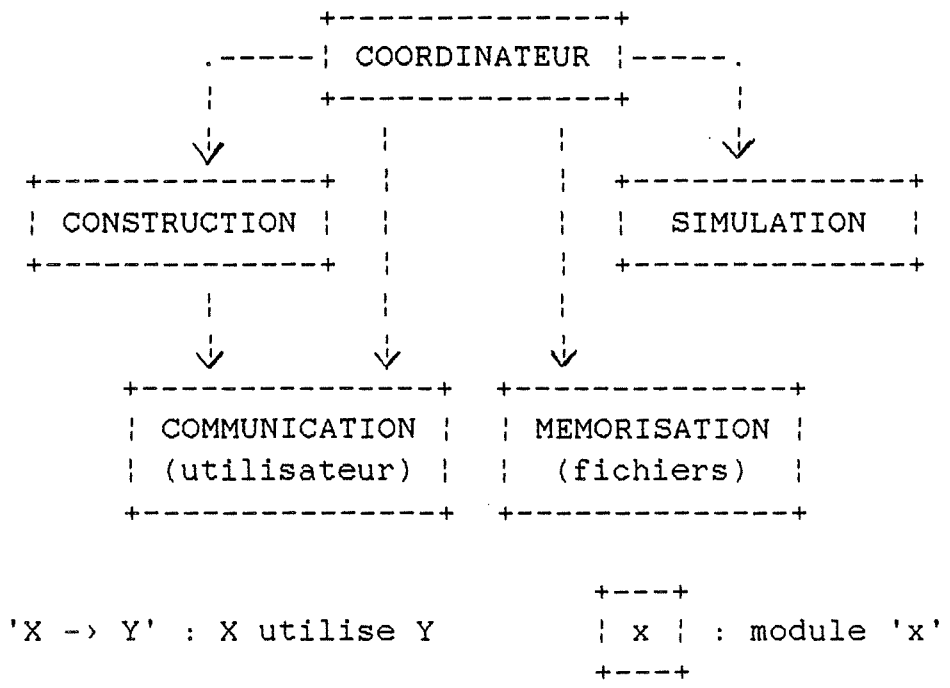


Figure 3.1

A partir de cette structure générale du didacticiel, nous allons définir le rôle de chacun des cinq modules.

3.1.1. Communication avec l'utilisateur

Pour l'instant, je ne définirai ce module que dans les grandes lignes. En effet, on peut facilement dénombrer les services généraux que l'on en attend mais sa définition détaillée est fortement influencée par son contexte d'implémentation.

De plus, si l'on veut garder un temps de réponse acceptable (surtout sur un micro-ordinateur), il me semble difficilement justifiable de définir une couche logicielle supplémentaire qui offrirait, dans tous les cas, une série de services standards.
Dans le cadre de ce chapitre, je me limiterai donc à citer les caractéristiques principales de ce module.

En ce qui concerne les sorties, il faut qu'il offre un ensemble de primitives permettant d'accéder de manière uniforme à des objets différents (par exemple : écran physique, fenêtre dans cet écran physique, écran logique, vue d'une partie d'un écran logique dans une fenêtre, représentation logique d'une fenêtre).
L'accès à ces objets se concrétise par leur création/modification (effacer, écrire, dessiner, supprimer l'objet, ...).

Ce module doit également permettre d'utiliser séparément les canaux d'entrées (clavier, souris,...) mais aussi proposer un canal d'entrée standard donnant une vue uniforme des canaux particuliers (par exemple rendre transparent le fait que la position du curseur soit modifiée par la frappe d'une touche au clavier ou le déplacement d'une souris).

Enfin, il est utile de disposer d'une gestion automatique de menus s'appuyant sur les services fournis par les primitives d'entrées-sorties mentionnées ci-dessus.

3.1.2. Mémorisation à long terme

Pour les mêmes raisons que précédemment (cfr 3.1.1), je n'évoquerai ici que les grandes fonctionnalités de ce module.

Celui-ci devra permettre de gérer le stockage d'un ensemble d'objets quelconques dont il ne connaît pas à priori la structure (si ce n'est que les éléments constitutifs de l'objet sont contigus : pas une structure de liste chaînée physiquement éparpillée dans la mémoire par exemple).
Les objets seront typés et tous les objets d'un même type auront la même taille.
On appellera *collection* un ensemble d'objets qui peuvent être de types différents.

Pour ces deux entités (objet, collection), on disposera d'un ensemble de primitives de manipulation (création, suppression, modification, accès) basées sur le principe d'élément courant avec une série de moyens pour changer d'élément courant.

A côté de ces primitives classiques, on prévoira des primitives d'accès partagé : le module de stockage ne connaît pas la structure des objets tandis que celui qui l'utilise ne connaît pas la structure d'une collection.
Partant de ce constat, il apparaît que la définition de primitives dont chacun des protagonistes ne connaît que la partie qui l'intéresse peut être fort utile : en effet, un tiers connaissant l'existence de deux modules (l'un gérant l'objet en mémoire et l'autre gérant le stockage) pourrait très bien sauver un objet dans une collection en ne connaissant absolument rien de lui (taille, structure, emplacement mémoire) ni de la manière dont il va être mémorisé (collection).
Pour cela il suffit de disposer des services suivants :

(1) une primitive d'initialisation d'un transfert partagé (de manière à ce que l'objet devienne l'objet courant de la collection courante)
(2) la référence de primitives de lecture/écriture d'un certain nombre de bytes définissant le contenu d'un objet (courant) dans une collection (courante).
(3) une primitive marquant la fin du transfert

Ces trois fonctionnalités sont les seules que doivent partager les différents modules. Pour le module gérant la représentation de l'objet en mémoire, il suffit même de connaître les spécifications des primitives du type (2).

3.1.3. Construction de la représentation du montage

Ce module sera chargé de la gestion du montage tel qu'il est vu par l'utilisateur.

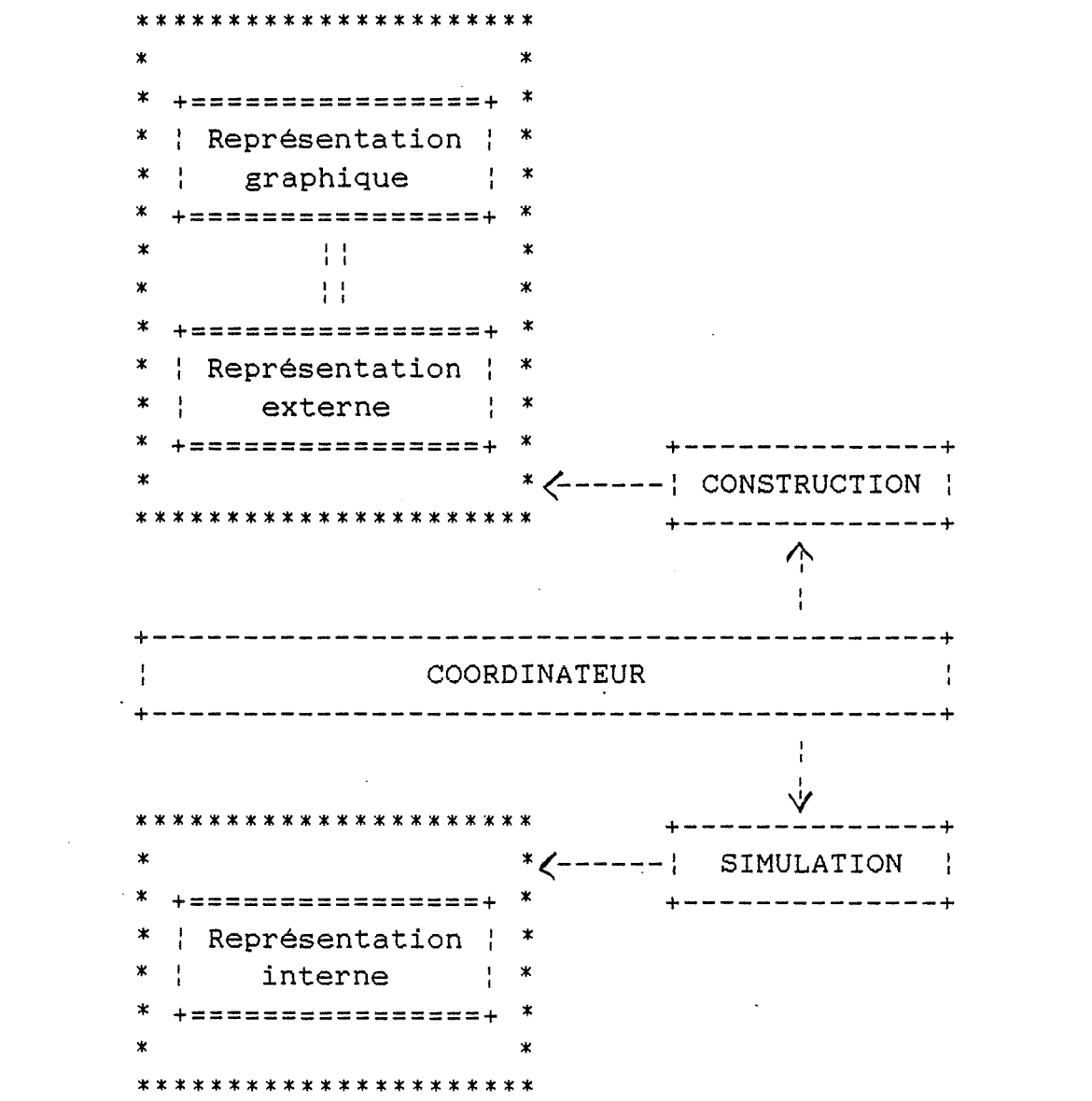


Figure 3.2

En se référant à la figure 3.2, on peut dire que ce module travaillera en parallèle sur deux types de données : la 'représentation graphique' du circuit (présentée à l'utilisateur) et la 'représentation externe' du circuit.
Cette dernière est définie comme la traduction logique du plan du circuit sous la forme d'une structure de données plus facilement manipulable d'un point de vue informatique.

Le rôle de ce module consistera à gérer en parallèle ces deux structures de données jouant ainsi le rôle d'interface entre l'utilisateur et le module coordinateur pour tout ce qui concerne le circuit proprement dit.

3.1.4. Simulation du comportement du montage

Le déroulement de l'étude du comportement du montage sera basé sur l'évolution dynamique (cfr stratégie de résolution) d'une 'représentation interne' du circuit.
Ce module ne sera pas en relation directe avec l'utilisateur.
Les communications avec le monde extérieur se limiteront à la définition de la 'représentation interne' et à des échanges d'informations concernant l'évolution du processus de simulation (aucune référence explicite à la représentation externe n'est nécessaire au processus de simulation).
Toute communication passera donc nécessairement par le module coordinateur.

Le rôle du module de simulation se limitera à la gestion dynamique de la 'représentation interne'. Nous le détaillerons plus loin dans le point consacré à la simulation du circuit.

3.1.5. Rôle du coordinateur

De la présentation succincte des modules précédents, il ressort que le module principal aura à assumer un rôle de chef d'orchestre.
Autrement dit, c'est lui qui devra faire le lien entre les différents services offerts par les autres modules.
Le prix à payer est que, comme un bon chef d'orchestre, le

module principal devra très bien connaître ses musiciens (voir figure 3.2).
C'est donc aux principaux d'entre-eux que nous allons maintenant nous intéresser, ce qui nous permettra, finalement, de tracer le portrait idéal du chef d'orchestre qu'il nous faut...

3.2. Construction du circuit

3.2.1. Que voit l'utilisateur ?

Il faut recréer pour l'utilisateur un environnement qui lui rappelle un laboratoire d'électricité.
Dans ce dernier, on trouve un certain nombre d'objets que l'on manipule. Ces objets peuvent se séparer en deux classes : la quincaillerie et les appareils de mesure.
Les premiers correspondent à des éléments qui permettent de construire un montage électrique, les seconds rendent possible l'observation du comportement de ce dernier.

3.2.2. Objets manipulés

La quincaillerie comprend les composants, les sources d'alimentation et des fils de cablage.
Quand tous ces éléments sont reliés entre eux en des points que nous avons appelés *noeuds*, les appareils de mesure servent à appréhender le comportement du circuit.
En disant cela, on tronque déjà la réalité, puisque le branchement des appareils de mesure fait souvent partie de l'étape de construction.
Néanmoins, dans le cadre de la simulation d'un laboratoire, il est possible de représenter directement le comportement d'un composant. C'est comme si, braquant son regard sur un composant, l'utilisateur pouvait "voir" directement ses caractéristiques électriques (courant, tension,...).

Les différents types d'objets manipulés seront donc des éléments constitutifs du circuit (résistances, capacités, selfs, fils de cablage, sources d'alimentation) reliés à des noeuds.

Pour cette application, seuls les schémas plans seront pris en considération : c'est à dire des schémas qui peuvent se dessiner dans un plan sans qu'il y ait d'intersection entre les branches (fils ou composants).

3.2.3. Rôle du module de construction

Le rôle de ce module consiste à traduire les directives de l'utilisateur en une représentation standard du circuit qui ne soit pas soumise à des contraintes physiques (nombre de composants pouvant être reliés à un noeud, représentation graphique du circuit dans un plan,....).
De plus, il faut pouvoir communiquer avec l'utilisateur. Cela suppose que l'on dispose d'un langage commun. Dans ce cas, il s'agit de la représentation du circuit par un schéma dont les éléments ont une certaine signification (un point représente un noeud; une droite, un fil,....).
Quand l'utilisateur désigne un élément du plan, son équivalent pour le didacticiel doit être facilement déterminé et inversement.

Un ordinateur arrive à dessiner mais il prend toujours beaucoup de temps pour reconnaître les éléments d'un dessin. Il manipule plus facilement des listes, des tableaux,... peu compréhensibles globalement par un être humain alors que ce dernier comprend mieux la signification d'un dessin.
Dès lors, il faut absolument disposer d'un outil à deux facettes, chacunes étant adaptées à l'interlocuteur avec lequel on veut communiquer.
C'est pourquoi il m'a semblé indispensable d'avoir une double représentation du circuit tel que l'utilisateur le voit, à savoir d'une part, un graphique représentant le plan du circuit et d'autre part, une structure de données, reflet fidèle de ce plan qui soit plus facilement manipulable par un programme d'ordinateur.

C'est cette structure que nous allons maintenant détailler, sachant qu'il doit y avoir une correspondance biunivoque entre les éléments des deux représentations.

3.2.4. Représentation externe du circuit

En observant les objets que nous allons devoir utiliser, il apparaît que les noeuds du circuit constituent des points communs entre les éléments de différentes natures. De plus, si l'on considère l'aspect d'un schéma de circuit électrique, on constate que les noeuds forment d'excellents points de repère.

Si l'on examine maintenant le support sur lequel les schémas sont dessinés, il faut admettre qu'il est de dimension finie et que ceux-ci deviennent difficiles à interpréter s'il y a beaucoup de croisements de fils de cablage. Or, n'oublions pas que nous sommes dans un contexte d'enseignement où la clarté doit nécessairement l'emporter sur la complexité.
Il est donc raisonnable d'envisager le support comme un ensemble de noeuds potentiels parmi lesquels l'utilisateur choisit ceux qui sont utiles à la représentation du circuit qu'il désire réaliser.
Nous considérerons donc que le support du schéma est constitué par un tableau de noeuds à deux dimensions (plan). Entre ceux-ci s'insèreront les différents composants du montage.

Quelles sont les informations utiles pour chaque noeud ?
D'une part, on retrouve les informations liées à leur localisation dans le plan. Celles-ci découlent implicitement de la position de leur représentation dans le tableau si l'on conçoit qu'ils sont distribués régulièrement dans l'espace du plan (tant horizontalement que verticalement).
D'autre part, il y a les informations concernant les composants qui y sont reliés.
Si l'on considère qu'à partir d'un noeud, il ne peut y avoir qu'un nombre fixe de connexions (8 branches par exemple, une tous les 45°), il suffit de connaître l'état de chaque direction possible. Une branche peut partir dans cette direction (à un coin, il y a moins de possibilités).
Si une branche existe dans une direction, quelle est sa nature (fil, composant, source) ?
Ces informations pourront éventuellement être complétées par une indication supplémentaire relative au fait que le noeud potentiel fasse ou non partie du circuit construit.

Abordons maintenant le schéma du côté des composants. Ceux-ci sont d'une certaine nature (résistance, capacité, self, source,...). Ils ont une valeur déterminée. Leur position dans le schéma est entièrement définie par les noeuds auxquels ils sont reliés si l'on admet que deux composants ne peuvent se superposer.

Si l'on rassemble maintenant les noeuds et les composants, on obtient une structure de données semblable à celle de la figure 3.3

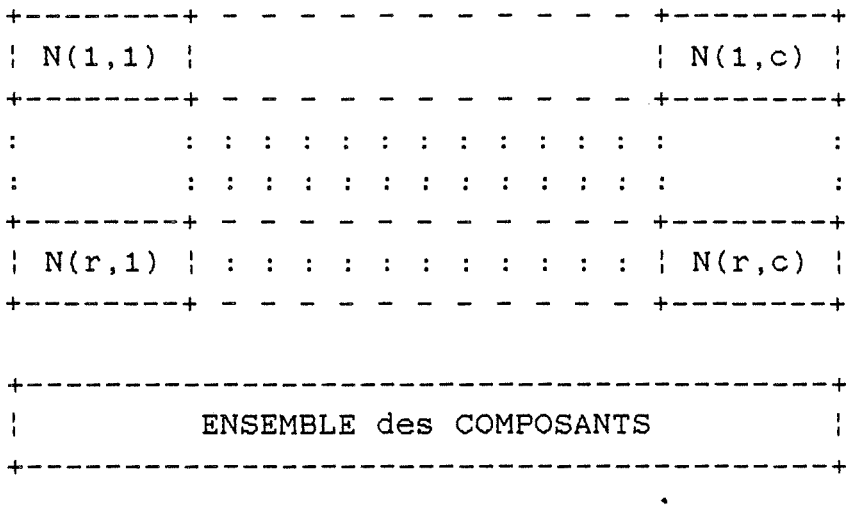


Figure 3.3

Dans celle-ci, les N(i,j) représentent les éléments du tableau de noeuds à deux dimensions.
Chacun d'entre eux comporte des indications "géographiques" (directions possibles pour les connexions, directions occupées, et pour chaque direction occupée : référence du composant correspondant).
Pour éviter les redondances inutiles, les composants sont regroupés (tableau, liste,...), chacun d'entre eux comportant les informations suivantes : références des noeuds auxquels il est connecté, type du composant, valeur, résultat de la simulation (courant, différence de potentiel,...)

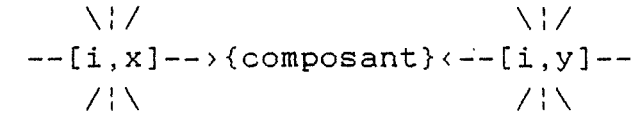


Figure 3.4

La figure 3.4 représente le schéma élémentaire d'un composant relié à deux noeuds ([i,x], [i,y]. Nous pouvons facilement imaginer les liens de correspondance directe qui existent entre ce graphique et la structure de données proposée à la figure 3.3.

Pour construire le circuit électrique, il faut disposer de ce que l'on pourrait appeler un éditeur_de_circuit. Voyons donc différentes fonctions que ce dernier doit pouvoir offrir.
On peut en distinguer trois types. Elles permettent soit d'ajouter des éléments à un schéma de base (une absence de schéma étant considérée comme un schéma particulier), soit de modifier des éléments de ce schéma, soit de détruire des éléments existants.
Ces trois types de fonctions vont correspondre à trois types d'actions possibles sur la représentation externe, qui seront directement répercutées sur le schéma du circuit montré à l'utilisateur ('représentation graphique).

Détaillons ci-dessous l'effet de ces types de fonctions sur la représentation externe du circuit.

1) Adjonction d'un noeud.

Si le noeud ne fait pas encore partie du schéma, il s'agit simplement de la notification du fait qu'il est sensé y appartenir.
Du côté utilisateur, cela doit se traduire par l'apparition d'un point sur le support de représentation.
Cette opération peut être implicite si, à un moment donné, on rajoute une branche au circuit alors qu'une de ces extrémités aboutit à un noeud qui n'appartient pas encore au schéma.
Remarquons également que le fait d'avoir défini une représentation discrète du support (tableau de noeuds), réduit le nombre des emplacements que peuvent occuper les noeuds dans le plan du schéma. Cette limite est absolument nécessaire si l'on veut simplifier les règles de correspondances entre la représentation externe et la représentation graphique.

2) Modification d'un noeud.

Ce type d'opération est entièrement implicite. En effet, la modification d'un noeud découle directement des modifications de connexions qui peuvent l'affecter.
La seule modification explicite que l'on pourrait à la rigueur accepter est le transfert des informations relatives à un noeud vers un autre noeud. Cela correspondrait au déplacement d'un noeud dans le schéma. Au point de vue de la représentation externe, ce serait assez simple mais au point de vue de la représentation graphique du schéma, cela constituerait une opération assez complexe. Néanmoins, on pourrait l'imaginer sous certaines conditions (la destination du noeud devrait pouvoir accueillir toutes les connexions en conservant les mêmes directions par exemple).

3) Suppression d'un noeud.

Il s'agit de supprimer un noeud du schéma ainsi que toutes les branches qui y sont connectées.

4) Adjonction d'un composant.

Cette opération consiste à rajouter une branche partant d'un noeud dans une certaine direction pour autant que cette dernière ne soit pas déjà occupée ou qu'elle ne soit pas interdite d'un point de vue graphique (hors des limites du plan, superposition avec une branche préexistante,...)

5) Modification d'un composant.

La modification d'un composant peut affecter sa signification physique (nature, valeur,...) mais aussi sa localisation (changer les noeuds auxquels le composant est relié). Cette dernière opération doit toutefois respecter un certain nombre de contraintes en ce qui concerne les nouvelles connexions (cfr Adjonction d'un composant)

6) Suppression d'un composant.

Cette opération se résume à la destruction d'une branche reliant deux noeuds.

Après avoir énuméré les six opérations de base de l'éditeur de circuit, notons que les opérations de modification d'emplacement de noeud ou de composant se résument à mémoriser les informations concernant l'élément à déplacer, pour ensuite le supprimer et enfin recréer ailleurs un nouvel élément grâce aux informations mémorisées. Cette manière de faire permet d'éviter les interférences qui peuvent survenir entre l'élément avant déplacement et l'élément après déplacement (voir surtout les contraintes d'ordre graphique).

Il y a cependant un problème que nous n'avons pas encore soulevé, il s'agit de la localisation des différents éléments du schéma dans le plan de leur représentation graphique.
Le fait que l'on ait représenté ce plan de manière discrète (tableau de noeuds) permet de résoudre directement ce problème si l'espace entre les noeuds est toujours le même tant horizontalement que verticalement.
Quant aux branches, elles sont localisées selon les cas : soit par leurs extrémités (les noeuds), soit par l'une d'entre elles (un noeud de départ), leur localisation étant alors déduite en fonction de leur forme et de leur orientation à partir du noeud de départ.

3.2.5. Architecture

Ayant défini les différents objets utilisés par ce module ainsi que leurs représentations, il est maintenant possible de dégager sa structure générale.

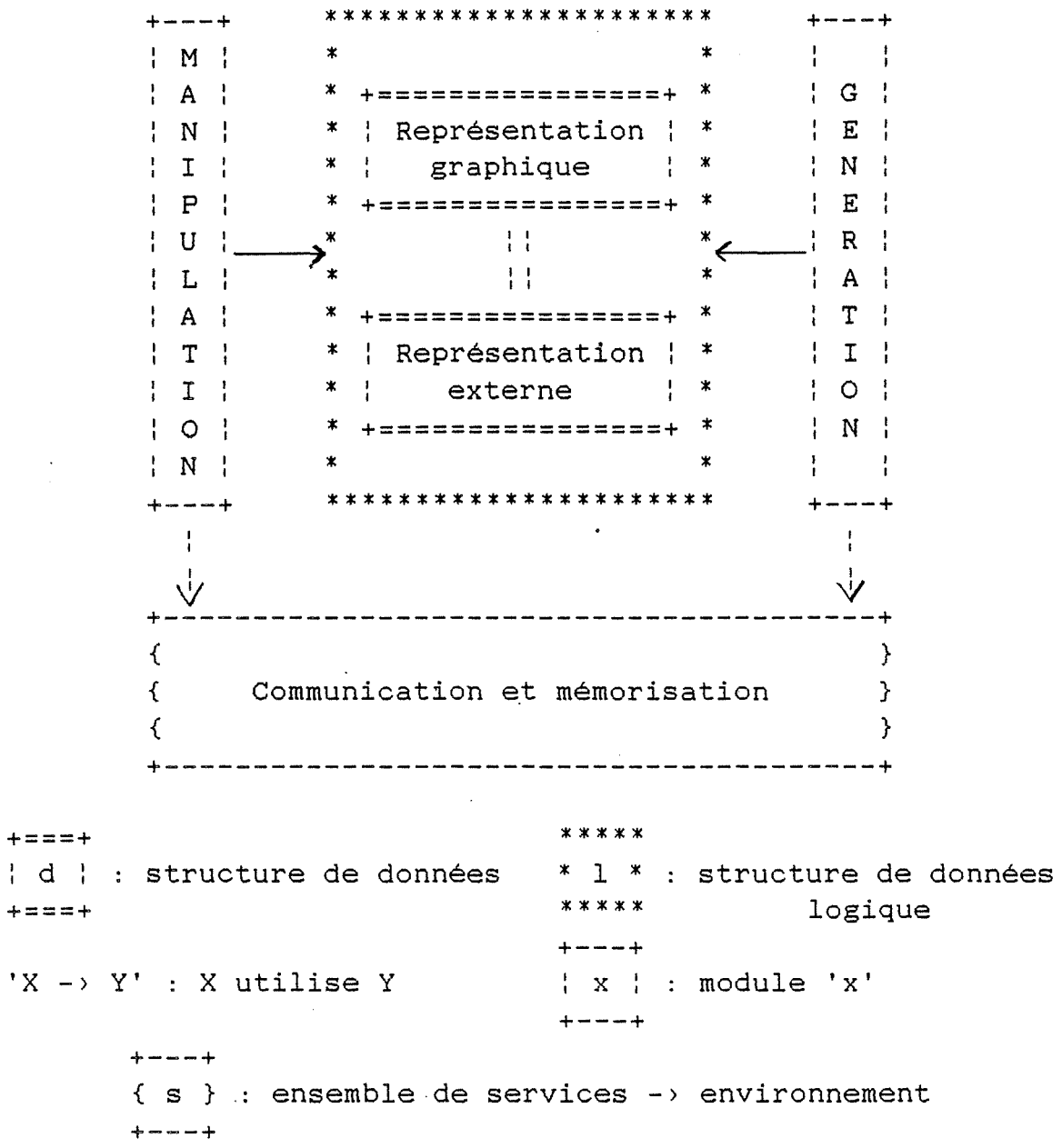


Figure 3.5

Ce module s'articulera autour de quatre composants principaux que l'on définira comme suit (cfr figure 3.5) :

- Un composant de 'représentation externe' dont le rôle est de fournir un moyen d'accéder à la structure de données sous la forme d'un ensemble de primitives.

- Un composant de 'représentation graphique' du circuit dont le rôle est d'offrir un certain nombre de primitives graphiques de haut niveau permettant de représenter facilement le schéma du circuit destiné à l'utilisateur.

- Un composant chargé de la génération du circuit (éditeur de circuit)

- Un composant permettant à l'utilisateur d'explorer le circuit en accédant aux résultats de la simulation (manipulation du circuit). Il est également chargé d'offrir un certain nombre de services visant à simplifier l'élaboration du dialogue avec l'utilisateur en ce qui concerne les échanges d'informations relatives au circuit après l'étape de construction .

Même s'il y a deux composants pour représenter le circuit électrique tel que l'utilisateur le voit (représentation graphique et représentation externe), il doit toujours y avoir une correspondance biunivoque entre les deux structures. L'utilité d'avoir deux composants réside dans le fait que cela permet d'effectuer un certain nombre d'opérations (tests, modifications,...) avant de les répercuter à l'écran.

3.3. Simulation du circuit

3.3.1. Stratégie de résolution

La stratégie de résolution proposée dans le deuxième chapitre peut se résumer en trois mots : réduction, propagation et superposition.
De plus, ce que nous voulons obtenir, c'est un simulateur dont les composants soient réutilisables dans d'autres contextes.

Des trois mots résumant la stratégie découle une première constatation : en tant qu'étape inverse de la réduction, la propagation n'est pas utile dans un contexte différent.
Quant à la superposition, son but est de permettre la réduction d'un circuit alimenté par une seule source, ce qui en fait un outil d'usage assez général.

De ces quelques considérations, il ressort qu'un simulateur permettant d'étudier le comportement d'un circuit alimenté par une source unique est un outil de base. Ceci constituera donc le rôle du module que nous allons étudier maintenant. Le problème de la superposition étant assez spécifique, nous le laisserons de côté momentanément. En effet, nous verrons dans le point consacré au module principal, que le principe de superposition peut jouer le rôle de trait d'union entre les représentations interne et externe du circuit.

3.3.2. Transformations

La stratégie de résolution se base sur les transformations d'éléments de circuit en composants équivalents.
Le problème posé par ce type d'approche est qu'il nécessite la mémorisation d'actions hétérogènes.
En effet, ces dernières portent sur un nombre variable d'éléments et ce, de manière différente.
La transformation parallèle réduit le nombre de branches du circuit entre deux noeuds.
La transformation série réduit à la fois le nombre de branches et le nombre de noeuds.
Enfin, la transformation étoile-triangle réduit le nombre

de noeuds, et quand on utilise sa version généralisée (plus de trois branches), elle augmente le nombre de branches.

Dans ce fouillis, nous pouvons cependant trouver un dénominateur commun : l'état du circuit.
Celui-ci est défini par la représentation des éléments touchés par une transformation avant et après cette dernière.
Si l'on considère l'état initial comme un circuit "vide", le premier changement consiste en la mise en place du circuit à étudier.

3.3.3. 'Représentation interne' du circuit

Du point précédent, on peut immédiatement tirer la conclusion qu'il est impossible de travailler directement sur la représentation externe. En effet, dans cette dernière, il peut y avoir plusieurs sources d'alimentation et il existe des contraintes liées à la représentation graphique qui disparaissent lorsque l'on se préoccupe uniquement du comportement du circuit (un fil de cablage entre deux noeuds sera transformé par la fusion des deux noeuds par exemple).

Comme j'ai opté pour une approche la plus modulaire possible, il devient impératif de mettre en oeuvre une 'représentation interne' du circuit.
En effet, les représentations interne et externe du circuit sont différentes par nature et par destination. Il devient donc utile d'en faire des entités séparées, ce qui permet leur utilisation dans un contexte plus large.

Pour les étapes de réduction et de propagation du résultat, toutes les transformations de la première devant être réversibles, il faut mémoriser un historique de l'évolution du circuit.
Néanmoins, ce dernier doit se présenter de manière homogène tout au long de chacune de ces étapes, il est donc nécessaire de disposer d'une représentation de son état courant à laquelle on puisse accéder directement.
De plus, tout changement de cet état courant devra être mémorisé avec un minimum de redondances.

Ce sont ces exigences qu'essaie de rencontrer la structure de données proposée à la figure 3.6. Cette dernière s'articule en trois composantes représentant les trois classes d'objets que le module manipule : les branches, les noeuds, jonctions des branches et la suite des transformations subies par le circuit (les étoiles font partie de la représentation des noeuds).

3.3.3.1. Aspects statiques de la structure de données

Nous allons maintenant discuter les différents éléments de cette structure de données en voyant si elle respecte bien les contraintes que l'on s'est donné plus haut, tout en permettant de représenter à la fois l'historique du circuit et son état courant.

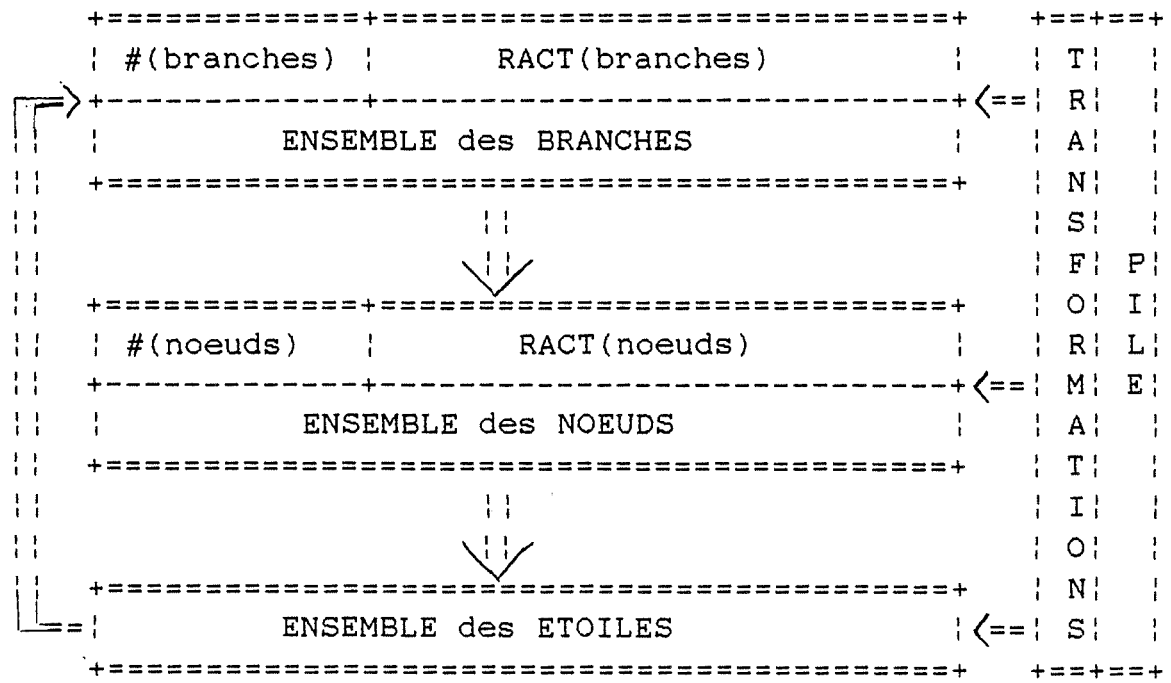


Figure 3.6

Je vais commencer par détailler succinctement les différents éléments constitutifs de cette structure.

On remarque tout d'abord trois composantes qui ont une double structure (branches, noeuds, transformations).
Celles-ci sont toutes trois constituées d'un ensemble d'éléments (tableau, liste,...) accompagné d'une structure décrivant l'état courant de l'ensemble. Cette dernière peut prendre l'un des deux aspects suivants : une suite de références préfixée par sa taille (cfr branches, noeuds) ou une pile de références (cfr transformations).
Reste la dernière composante (étoiles) qui est un ensemble d'éléments (tableau, liste,...) de taille variable dont chacun est la représentation d'une étoile, c'est-à-dire une suite de références de branches (préfixée ou non par le nombre de références qu'elle comporte). Chaque noeud contiendra la référence de l'étoile dont il est le centre.

Deux raisons m'ont incité à diviser la représentation des noeuds en deux composantes (noeuds, étoiles).
La seule partie variable de la représentation d'un noeud est constituée par les références des branches qui y sont reliées. Or, on accède bien plus souvent aux informations de taille fixe qu'à cette information de taille variable (la suite de références). Il était donc logique de les séparer puisque la gestion de l'accès à des éléments de taille variable est toujours plus coûteuse que l'accès à un vecteur d'éléments de taille fixe.
La seconde raison, plus importante à mes yeux que la première, est liée à l'aspect dynamique de la structure.
Nous verrons plus loin qu'une partie importante des modifications apportées par une transformation affectera la représentation des noeuds du point de vue des branches qui y sont connectées. Comme ce fragment d'information (étoile) est partagé par plusieurs composants (noeuds, transformations), il est utile de pouvoir le manipuler sous forme de référence plutôt qu'en le duplicant.

3.3.3.2. Aspects dynamiques de la structure de données

Présentée comme cela, cette structure de données peut sembler un peu difficile à comprendre. C'est pourquoi nous allons l'examiner en train de "vivre" en partant de son état initial et en voyant comment chaque transformation du

circuit s'y inscrit.
Après cela, nous en déduirons les informations mémorisées par chaque élément de la structure.

Comme je l'ai défini plus haut, l'état initial est celui de l'absence totale de circuit (structure de données vide).
La première transformation consiste à charger la représentation de ce dernier. A ce niveau, il faut observer deux points particuliers : la source d'alimentation (branche) et les deux noeuds auxquels cette dernière est reliée. En effet, ces éléments ont un statut un peu différent parce qu'ils doivent être protégés de toute destruction lors de l'étape de réduction.
Pour le reste, il s'agit de remplir la structure avec les éléments constitutifs du circuit. Je ne détaille pas ce remplissage puisque nous ne disposons pas encore de toutes les informations nécessaires concernant la structure.
Néanmoins, constatons qu'à cette étape, la pile des transformations est vide alors que les trois autres éléments représentent le circuit de départ.
Nous allons voir maintenant quel est l'effet de chaque type de transformations pour nous intéresser enfin à la phase d'initialisation qui permettra de faire le lien avec la 'représentation externe'.

Pour exprimer graphiquement l'évolution de la structure de données, nous utiliserons le formalisme suivant :

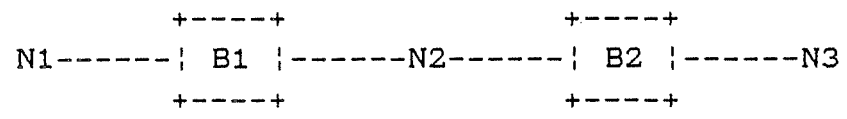
```
B           : Branche
N           : Noeud
E           : Etoile
Pour X=(B,N,E),
Xi          : élément i de type X
Xi'         : nouvel élément i' de type X
X(Y)        : l'élément X contient la référence de Y
A->B        : il existe un pointeur de A vers B
              (équivalent à A(B))
S1 ==> S2 :  S1 est l'ensemble des éléments pris
                avant la transformation
             S2 est l'ensemble des éléments pris
                après la transformation
X           : fragment non actif de la structure de données
```

Transformation Série

Cette transformation consiste à trouver des noeuds d'où ne partent que deux branches pour ensuite opérer la fusion de ces dernières.

Pour ce faire, on parcourt l'ensemble des noeuds actifs jusqu'à ce que l'on en repère un ayant les caractéristiques nécessaires.
L'ayant trouvé, il faut d'abord vérifier s'il ne s'agit pas des bornes de l'alimentation.

AVANT



B1(N1,N2) B2(N2,N3)
N1->E1(B1,...) N2->E2(B1,B2) N3->E3(B2,...)

TRANSFORMATION

N2 ==> B, E1,E3

APRES

B1(N1,N2) B2(N2,N3) N2->E2(B1,B2) *E1(B1,...) E2(B2,...)*
B(N1,N3)
N1->E1'(B,...) N3->E3'(B,...)

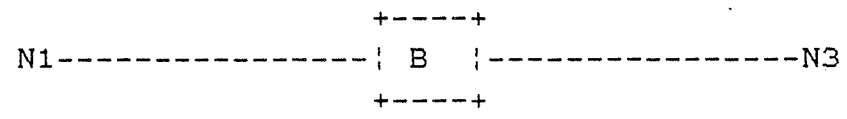


Figure 3.7

On applique alors la transformation qui a pour effet de supprimer le noeud de la liste des noeuds actifs ainsi que les deux branches impliquées (liste des branches actives), pour ensuite générer la branche équivalente aux deux branches supprimées.

Voyons les caractéristiques des deux états (avant et après tranformation) en nous référant à la figure 3.7.

Pour ce qui est de l'état avant transformation, la référence du noeud supprimé (N2) est mémorisée (celui-ci contient la référence de l'étoile (E2) et donc des deux branches (B1,B2)).
Il faut retenir que par cette transformation, le nombre de branches actives ainsi que le nombre de noeuds actifs diminuent chacun de une unité.

Enfin pour l'état après transformation, on retiendra la référence de la branche (B) générée (qui contient la référence des deux noeuds (N1,N3) auxquels elle est reliée).
En ce qui concerne ces deux noeuds, il y aura modification des références des branches qui leur sont connectées. Il faudra donc générer deux nouvelles étoiles (E1',E3') tenant compte des changements intervenus, mais aussi mémoriser la référence des deux anciennes étoiles correspondantes (E1,E3).

Ceci entraîne une certaine redondance puisqu'il suffirait de modifier la valeur d'un élément dans chacune des étoiles impliquées. Cette mesure d'économie ne s'impose que si l'on privilégie la réduction de l'encombrement mémoire sur la simplicité de l'algorithme de cette transformation et surtout de celle de la transformation inverse correspondante.

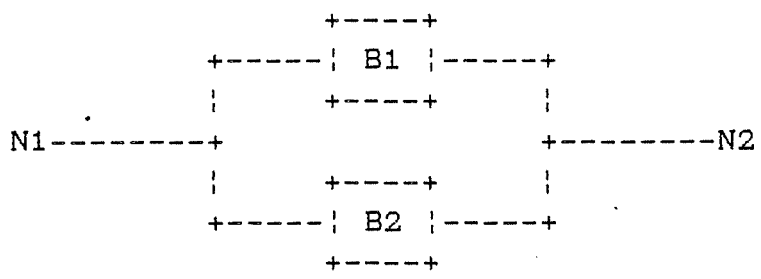
Transformation Parallèle

Cette transformation consiste à trouver des couples de branches partageant les mêmes noeuds.

Il suffit pour cela de parcourir la liste des branches actives en les comparant deux à deux du point de vue du couple de références des noeuds auxquels elles sont reliées. Un couple de références (R11,R12) sera identique à un autre (R21,R22) dans deux cas :
- si (R11,R12) = (R21,R22)
- si (R11,R12) = (R22,R21)

Ayant repéré un tel couple de branches, il faut d'abord vérifier qu'aucune d'entre elles n'est la source d'alimentation. Ensuite seulement, on peut générer la transformation (branche équivalente).
Etudions de nouveau les deux états liés à cette transformation (voir figure 3.8).

AVANT



B1(N1,N2) B2(N2,N3)
N1->E1(B1,B2,...) N2->E2(B1,B2,...)

TRANSFORMATION

B1,B2,E1,E2 ==> B

APRES

B1(N1,N2) B2(N2,N3) E1(B1,B2...) E2(B1,B2...)
B(N1,N2)
N1->E1'(B,...) N2->E2'(B,...)

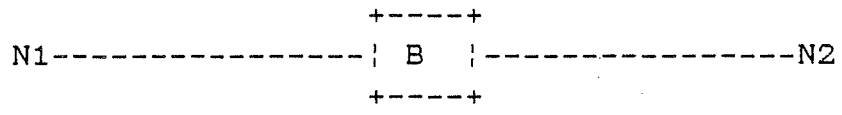


Figure 3.8

Pour ce qui est de l'état avant transformation, il suffit de mémoriser les références des branches (B1,B2) supprimées et celles des étoiles relatives aux deux noeuds (E1,E2) impliqués dans la transformation (ici on ne peut plus reprendre la même structure puisqu'il y a diminution du nombre de branches pour chaque étoile).

Il faut retenir que la transformation a diminué de une unité le nombre de branches actives.

Enfin pour l'état <u>après transformation</u>, on retiendra la référence de la branche (B) générée (qui contient la référence des deux noeuds (N1,N2) auxquels elle est reliée).
En ce qui concerne ces deux noeuds, il y aura modification des références des branches qui leur sont connectées, il faudra donc générer deux nouvelles étoiles (E1',E2') tenant compte des changements intervenus. Cette fois, il est inutile de mémoriser leur référence qui est accessible par le chemin branche-noeuds-étoiles.

Avant d'aborder la transformation suivante, on peut remarquer que les deux précédentes ont un point commun : le parcours d'un ensemble d'éléments (branches, noeuds) parmi lesquels il faut distinguer un intrus (la source d'alimentation ou ses bornes). Pour éviter cette vérification à postériori, les intrus figureront en tête de la suite des références des éléments actifs. En effet, dans ce cas, ils seront éliminés d'emblée si le parcours de l'ensemble ne doit commencer que par les références suivantes.

<u>Transformation Etoile-triangle</u>

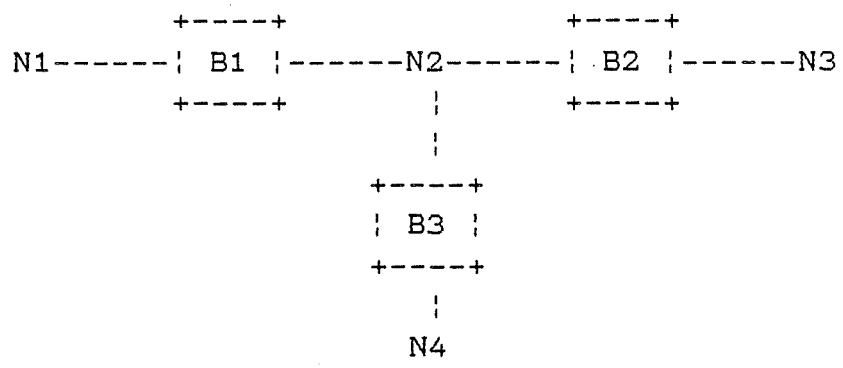
De nouveau, pour cette transformation, il faut parcourir l'ensemble des références des noeuds actifs en évitant les bornes de la source d'alimentation jusqu'à ce que l'on trouve un noeud d'où partent trois branches.

Reportons-nous à la figure 3.9 pour étudier les modifications de la structure de données.

Pour mémoriser l'état <u>avant transformation</u>, il suffit de retenir la référence (N2) du centre de l'étoile (et donc des trois branches qui y sont reliées) ainsi que les références des trois étoiles (E1, E2, E3) auxquelles aboutissent ces trois branches.

En ce qui concerne l'état <u>après transformation</u>, il faut mémoriser les références des trois branches générées (B13, B14, B34) aini que celles des trois nouvelles étoiles (N1, N2, N3) reliées par ces trois nouvelles branches.

AVANT



B1(N1,N2) B2(N2,N3) B3(N4,N2)
N2->E2(B1,B2,B3)
N1->E1(B1,...) N3->E3(B2,...) N4->E4(B4,...)

TRANSFORMATION

N2, E1, E3, E4 ==> B13, B14, B34, N1, N3, N4

APRES

B1(N1,N2) B2(N2,N3) B3->E2(N4,N3)
N2->E2(B1,B2,B3) E1(B1,...) E3(B2,...) E4(B3,...)
B13(N1,N3) B14(N1,N4) B34(N4,N3)
N1->E1'(B13,B14,...) N3->E3'(B13,B34...) N4(B14,B34,...)

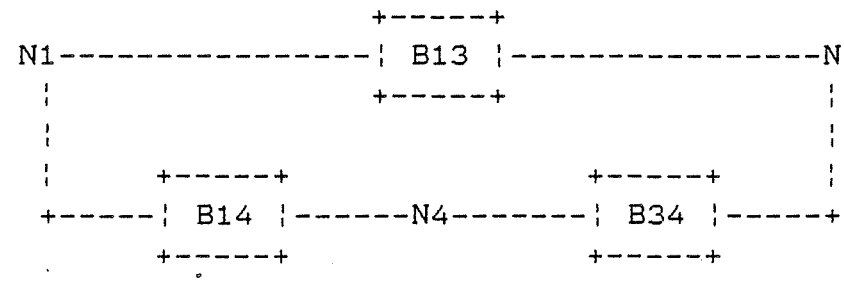


Figure 3.9

On retiendra également que la transformation a diminué d'une unité le nombre de noeuds actifs. Le nombre de branches actives reste inchangé dans ce cas (-3+3=0).

Si on voulait utiliser des transformations de réduction d'étoile de plus de trois branches, la seule différence

résiderait dans le fait que le nombre de branches générées serait plus élevé que le nombre de branches remplacées (cfr principes théoriques).

Nous disposons maintenant d'assez d'éléments pour détailler les composants de la structure de données. Nous n'avons cependant pas encore parlé de la découpe du circuit en étages, ni de l'influence de cette découpe sur le déroulement de la réduction.
Remarquons simplement que les seules informations liées à ces deux aspects sont d'une part, le sens du courant traversant chaque branche et d'autre part, l'identifiant de l'étage auquel appartient chaque noeud.

Voyons donc de quoi se composent les éléments de la structure de données.

L'ensemble des branches est une suite d'éléments de longueur fixe reprenant les informations suivantes : référence de deux noeuds, sens du courant du premier noeud vers le second, impédance, courant, différence de potentiel. Pour la source, on ignore la partie impédance. Ceci est sans conséquence puisque par convention, on placera la référence de celle-ci en tête de la suite des références des branches actives (ce qui permet de la situer de manière univoque).

L'ensemble des noeuds est également une suite d'éléments de longueur fixe reprenant les informations suivantes : la référence de l'étoile dont il est le centre, le nombre de branches que comporte cette étoile, l'identifiant de l'étage auquel il appartient.

La pile des transformations est une pile d'éléments de taille variable. Chacun d'entre eux comprendra, en règle générale, les informations suivantes : le type de transformation, les références des éléments modifiés par la transformation (état avant transformation) et les références générées par la transformation (état après transformation).
Un moyen simple de gérer une telle pile est de la diviser en deux parties : une pile de stockage des éléments de taille variable et une pile de références vers cette zone de stockage (éléments de taille fixe).

Ce moyen permet un parcours aisé de la pile une fois celle-ci constituée : ce qui est essentiel ici puisque l'étape de propagation sera guidée par une lecture de la pile dans le sens inverse de sa construction.

3.3.3.3. Etape d'initialisation de la structure de données

Il reste maintenant deux étapes à définir, le chargement de la structure et la découpe en étages.
Je propose ici de développer une approche qui permet de franchir ces deux étapes en même temps.
Il s'agit d'une adaptation d'un processus d'exploration d'un graphe en largeur d'abord.

En évoquant la structure de la 'représentation externe' du schéma, on songe que lors de la simulation, on doit être capable d'y revenir à tout moment. On remarque que les seuls éléments communs aux deux structures de données sont les branches (cfr les 'composants' de la structure externe).

Voyons maintenant quelques définitions qui nous seront utiles.

Un étage (cfr 2.3.1) est un ensemble de noeuds reliés par au moins une branche à l'étage précédent. Le premier étage est constitué des bornes de l'alimentation.

La génération d'un noeud consiste à créer un élément de l'ensemble des noeuds (cfr figure 3.6) ainsi qu'une nouvelle étoile dont la première référence indique la "première" branche qui rattache le centre de l'étoile à l'étage précédent (le nombre de branches formant l'étoile doit être connu mais les références de ces branches ne doivent pas nécessairement être définies lors de cette action).

Une branche sera dite active une fois que les deux noeuds auxquels elle est rattachée auront été générés.

La génération d'une branche consiste à créer un élément de l'ensemble des branches dont la structure est indéterminée sauf en ce qui concerne ses caractéristiques électriques.

Lors du processus d'initialisation, une branche ou un

noeud seront toujours identifiés par leur numéro d'ordre de génération.

Nous pouvons maintenant détailler le processus d'initialisation. Après avoir défini une structure de données vide, on procède à la génération de toutes les branches du circuit (posons qu'il en existe *B*) en commençant par l'alimentation.

La construction du premier étage consiste à générer les deux noeuds correspondant aux deux bornes de la source d'alimentation en commençant par la borne positive. La branche alimentation est donc active. Les étoiles des deux noeuds sont créées mais ne comportent que la référence de l'alimentation.

Les conditions générales sont établies et le processus ne s'arrêtera que lorsque les *B* branches seront actives.

Passons maintenant au traitement que l'on fait subir aux noeuds constituant un étage.
Pour chacun d'entre-eux, on complète la liste des références des branches reliées à l'étoile dont ils sont le centre.
Ayant une liste de références entièrement déterminée, on s'intéresse à chacune des branches en particulier.
S'il s'agit d'une branche qui n'est pas encore active, on détermine le second noeud auquel elle est reliée.
On procède à la génération de ce noeud s'il n'existait pas encore et dans ce cas, il s'agit d'un noeud appartenant à l'étage suivant.
On ajoute la référence de la branche à l'étoile dont le second noeud est le centre.
Comme les deux noeuds auxquels la branche est reliée sont maintenant déterminés, cette dernière devient active.

En résumé, l'algorithme qui découle de ce processus aura la forme suivante (algorithme 3.1).

```
INITIALISATION de la structure de données
GENERATION des B branches
CONSTRUCTION DU PREMIER ETAGE
tant que ((nombre de références de branches actives) < B)
  ¦ pour tous les noeuds de l'ETAGE COURANT
  ¦    ¦ Bref:=Première référence de l'étoile
  ¦    ¦ pour chaque ((branche de l'étoile) <> Bref)
  ¦    ¦    ¦ si (la branche n'est pas encore active)
  ¦    ¦    ¦    alors ¦ GENERER 1 noeud de l'étage suivant
  ¦    ¦    ¦          ¦ la branche DEVIENT ACTIVE
  ¦ PASSER à l'ETAGE SUIVANT
```

Algorithme 3.1

Dans ce qui précède, je n'ai pas fait mention de la détermination du sens du courant.
Celle-ci découle automatiquement du processus de génération de la représentation interne du circuit.
En effet, pour chaque noeud, il suffit de retenir quelle est la borne de l'alimentation la plus proche.
Cela se détermine aisément en notant si le noeud a été atteint par une branche positive ou négative.
C'est trivial pour les deux bornes de l'alimentation, il suffit dès lors de répercuter itérativement cette information lors de l'exploration du circuit.
Une branche comprend toujours la référence de deux noeuds, il suffit d'adopter une convention qui dit que la première référence est celle de la borne positive (sens conventionnel du courant).
Par extension, un noeud sera positif ou négatif en fonction du premier noeud auquel il est relié à l'étage précédent.
Lorsque l'on rend une branche active, on sait donc (en fonction du type de noeud) où l'on doit placer la référence du noeud. S'il y a conflit (la place est déjà occupée), on le résoud en prenant la place restée libre.
Comme on est parti de l'alimentation, le fait qu'un noeud ait déjà "pris la place convoitée" indique que le chemin entre lui et la source d'alimentation est plus court et par convention, nous avons décider de choisir le sens qui, chronologiquement, aura été déterminé le premier au cours du processus.

3.3.4. Structure du module

Ce module s'articulera donc autour de la représentation interne du circuit sous la forme de trois grands composants.
Le premier servira à construire cette représentation à l'aide d'informations venues de l'extérieur (du module principal par exemple). Le second aura pour tâche de réduire le circuit en son équivalent élémentaire. Enfin le troisième sera chargé de propager les informations recueillies au cours de la réduction.
Ce dernier composant n'a pas été détaillé parce qu'il utilise la suite inverse des transformations de réduction pour propager les informations concernant le comportement des composants équivalents par application directe des formules mises en évidence dans le chapitre deux.
Cette lecture inverse de la pile des transformations n'apportera dans la structure de données que les changements suivants :

- Mise à jour des suites de références de branches et noeuds actifs.
- Modification des références des étoiles pour les noeuds actifs.
- Déduction du courant et de la différence de potentiel pour chaque branche de la structure

3.4. De l'interface à la simulation

Ayant défini les possibilités des musiciens, il reste maintement à trouver le profil du chef d'orchestre qui parviendra à transformer cette boîte à outils en didacticiel.

En reprenant les modules principaux dont on dispose, on obtient la structure présentée à la figure 3.10

On remarque immédiatement que le rôle du module coordinateur est loin d'être négligeable. C'est le prix à payer quand on veut utiliser des outils d'un usage assez général. Ceci implique un certain savoir-faire de manière à en tirer le maximum et ce, dans des circonstances différentes.

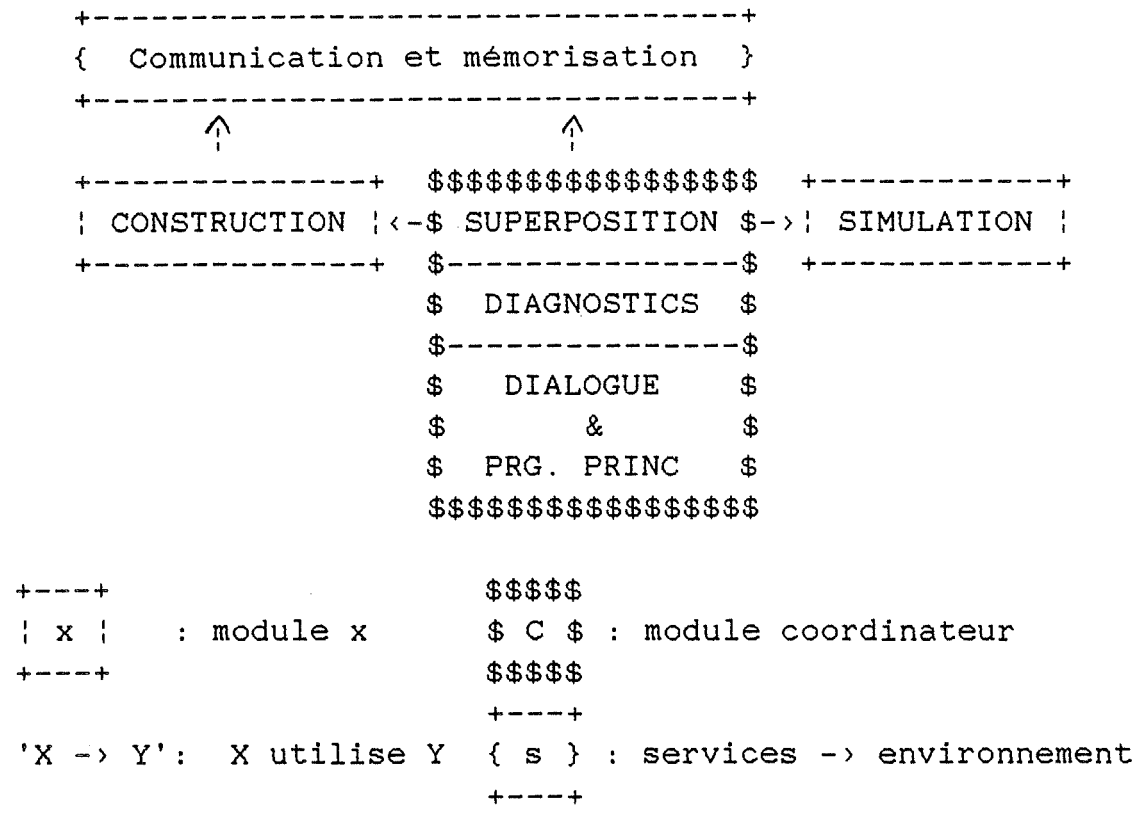


figure 3.10

Ce que nous demandons à notre didacticiel, c'est de simuler un laboratoire d'électricité à l'aide des différents modules présentés ci-dessus.

Pour arriver à ce résultat, il reste un certain nombre de choses à réaliser.

Dans la figure 3.10, le module coordinateur est présenté comme une entité logique ayant trois composantes principales (programme principal, diagnostic et superposition). En fonction de la taille de l'application finale, il est certainement intéressant de créer deux modules avec les composants de superposition et diagnostic de manière à pouvoir les récupérer.

Le programme principal aura à diriger le dialogue avec l'utilisateur et à traduire les ordres de ce dernier par un certain nombre de tâches qu'il effectuera à l'aide des outils dont il dispose.

Le composant émettant les diagnostics sera, quant à lui, chargé d'un dialogue particulier avec l'utilisateur : celui qui traite des informations concernant l'évolution de la simulation d'un circuit particulier.

Enfin, le composant de surperposition devra concilier les profils d'utilisation des outils de construction et de simulation.
Le module de construction permet de réaliser des circuits comportant plusieurs sources alors que le module de simulation ne peut étudier que des circuits comportant une seule source. Il faut donc superposer plusieurs études de circuits à source unique pour obtenir une image du comportement des circuits construits. Inversement, le composant chargé de la superposition aura aussi à construire la 'représentation interne' du circuit à partir de sa 'représentation externe'. En effet, il est tout à fait désigné pour savoir ce qu'il faut faire des autres sources pendant que l'on considère le comportement du circuit alimenté par une seule d'entre elles.

Cette structure est un bon compromis, me semble-t-il, entre d'une part, le fait que l'on veut développer des outils modulaires (indépendants) qui gardent une approche assez générale pour pouvoir être réutilisés dans d'autres contextes et d'autre part, la contrainte de spécialisation que nous avons évoquée dans le premier chapitre.

CHAPITRE IV

Choix d'implémentation

La présentation des modules et des structures de données principales, dans le chapitre précédent, permet maintenant de consacrer celui-ci à la discussion de quelques choix d'implémentation.
Dans un premier point, le contexte sera défini tant du point de vue de l'environnement d'exécution des différents modules, que de celui des outils utilisés pour les concevoir.
A partir d'options générales, les modules physiques correspondant à chaque module logique du chapitre précédent pourront être définis.
Cette définition s'accompagnera de la présentation des principaux choix d'implémentation qui s'y rattachent.

4.1. Contexte

4.1.1. Matériel

Dans le domaine de l'EAO, un élément déterminant du contexte est le type d'ordinateur sur lequel on décide de développer les applications. En effet, les contraintes économiques étant ce qu'elles sont, le matériel dont dispose l'enseignement est généralement peu coûteux et donc du style micro-ordinateur. Heureusement, ces dernières années, les possibilités de celui-ci se sont accrues de manière spectaculaire et l'on peut même dire que, dans un contexte d'enseignement, l'atout principal d'un micro-ordinateur est sa grande souplesse d'utilisation.
Pour le moment, les PC de la veine des compatibles IBM-PC sont devenus un "standard" de la micro-informatique bon marché permettant d'atteindre des performances acceptables. C'est pourquoi j'ai décidé d'utiliser ce type de matériel, en me basant sur une configuration relativement simple (pas de mode écran nécessitant l'emploi de moniteur très évolué, pas de mémoire étendue, pas de coprocesseur mathématique obligatoire,.....)
Bien entendu, comme il s'agit d'un choix lié au matériel, on peut dire qu'à l'heure où j'écris ces lignes, il est déjà dépassé. Cependant, en ce qui concerne l'informatique dite "familiale", on peut penser qu'il a encore quelques beaux jours devant lui... Or, dans l'enseignement, c'est souvent à ce niveau d'informatisation que l'on se trouve.

4.1.2. Langage

Une fois ce décor planté, reste à choisir les outils de développement adéquats.

Une première contrainte est liée au fait que le résultat de l'implémentation doit rester relativement "lisible", il faut donc opter pour un langage assez répandu qui ne soit pas trop rébarbatif et qui soit aussi "portable" que possible. Dans ce domaine, le pascal est un bon compromis : tout en étant un langage structuré, il est de plus en plus répandu dans le monde des non-professionnels.
En ce qui concerne les compilateurs adaptés à l'environ-

nement MS-DOS, le turbo pascal de Borland constitue, me semble-t-il, un excellent choix, surtout à partir de sa quatrième version qui permet d'avoir une approche de développement vraiment "modulaire".

Malheureusement, on ne peut pas toujours utiliser un langage de haut niveau.
En effet, pour implémenter des fonctions de bas niveau (proches du hardware), il est parfois indispensable de pouvoir accéder directement aux ressources élémentaires de la machine, ce que les langages de haut niveau permettent mal, voir même pas du tout.
L'assembleur reste, dans ce cas, un outil de développement incontournable pour certaines parties limitées qui sont souvent invisibles pour un utilisateur n'employant que les fonctions de haut niveau que l'on aura développées.

4.2. Choix d'ordre général

4.2.1. Complexité des circuits

Le choix essentiel que j'ai posé, consiste à limiter la portée de la stratégie de résolution en ne permettant pas la réduction d'étoiles comportant plus de trois branches.
Si l'on se réfère au point 5.3 du second chapitre, on constate que ce choix permet de réduire la taille de la représentation interne du circuit (surtout d'un point de vue dynamique).
Cela présente l'avantage d'obtenir assez rapidement la solution de problèmes pas trop complexes.
De plus, comme je l'ai signalé dans le chapitre deux, le niveau moyen des exercices proposés dans un cours d'électricité de l'enseignement secondaire peut, dans la majorité des cas, être résolu par une méthode aussi limitée.
Quant aux problèmes plus complexes, leur résolution (même par d'autres méthodes) ne requiert pas d'autres connaissances que celles qui permettent de résoudre des problèmes plus simples. Dans ce cas, la difficulté est d'ordre quantitatif uniquement. Or, je ne crois pas, pour ma part, que le but d'un cours d'électricité réside dans cet aspect quantitatif.

4.2.2. Boîte à outils

Les exigences qui ont été posées tout au long des points précédents peuvent se résumer comme suit :

- approche modulaire
  (interdépendance faible entre les modules, services utilisables en connaissant uniquement l'interface du module : ces deux facteurs augmentent la "portabilité" de l'effort de développement des modules de haut niveau).
- "réutilisabilité"
  (les services offerts par les différents modules doivent être d'intérêt assez général pour pouvoir être utilisés dans d'autres contextes)
- produit fini
  (l'ensemble des modules doit pouvoir simuler un certain nombre de fonctionnalités d'un laboratoire d'électricité directement utilisables par l'élève).

Cela me conduit à développer une boîte à outils intégrés plutôt qu'un ensemble de modules spécifiquement destinés à une seule implémentation.
Ce choix est lourd de conséquences puisqu'il demande de développer des outils qui offrent plus de possibilités que celles strictement nécessaires à l'implémentation d'UN simulateur particulier.

4.2.3. Méthode de présentation

J'ai constaté que ce qui manque souvent à la description d'un logiciel, c'est l'énoncé des idées sous-jacentes qui ont guidé toute la démarche de sa conception.
La plupart du temps, on ne trouve qu'une longue suite aride de spécifications ou une version plus ou moins bien documentée de la traduction en code de ces dernières.

Il me semble pourtant intéressant d'avoir une démarche permettant d'appréhender le logiciel à différents niveaux de détails :

- une présentation générale des idées et des options qui ont guidé l'implémentation des différents modules
- la spécification des différents modules vu de l'extérieur (interface des modules)
- la manière dont les modules ont été développés (code source du logiciel)

C'est pourquoi je me limiterai dans le point suivant à présenter succinctement les différentes unités qui constituent le logiciel, réservant des annexes à la description de leur *interface* (cfr *units* du turbo 4).

4.3. Découpe en unités

Nous allons maintenant découper le didacticiel en "unités" (*units* du turbo 4). Ceci correspondra à une découpe en modules physiques.

4.3.1. Les fonctions de bas niveau

Ce n'est pas pour le plaisir de refaire le monde que j'ai consacré une partie non négligeable de l'effort de développement à la construction d'outils permettant d'accéder à un certain nombre de ressources de base.
Cela offre d'énormes avantages qui permettent de justifier le prix de l'effort consenti.
L'avantage majeur réside dans le fait que l'utilisation de ces services élémentaires rend les modules de haut niveau largement indépendants de l'environnement (MS-DOS, Turbo Pascal, PC).
Bon nombre de primitives existaient déjà sous l'une ou l'autre forme, mais elles présentaient assez de défauts pour justifier leur réécriture.
Je vais maintenant passer brièvement en revue les différents types d'outils qui ont été construits.

4.3.1.1. Hardware

Au niveau de l'accès des ressources offertes par le hardware (BIOS surtout), il fallait garantir une sauvegarde du contexte (registres et stack) qui soit à toute épreuve.
En effet, la majorité des primitives de ce type se contentent de sauver les registres sur le stack sans sauver l'état de ce dernier. Or, certaines primitives ne garantissent pas toujours la restitution exacte de la position de ce dernier (restitution ou non des flags, par exemple,...).

De plus, dans le cadre du turbo pascal, l'accès à une routine de gestion d'interruption interne au programme courant peut, dans certains cas, se terminer par un arrêt du système parce que le compilateur se base sur le fait que les routines du BIOS ne sont pas réentrantes pour se permettre de les générer d'une manière qui ne le soit pas non plus. C'est pourquoi il m'a semblé opportun de redéfinir un certain nombre de primitives de base qui permettent de générer des appels système dans les meilleures conditions possibles. Cela m'a permis également de simplifier l'utilisation de certaines ressources telles que l'écran, le clavier et les disques.

Ces primitives se trouvent dans les unités BDOS et UBREAK.

4.3.1.2. Fichiers

Pour gérer les fichiers et la structure logique du disque, il faut recourir à un certain nombre de services du MS-DOS dont la gestion est assez lourde (génération d'une interruption). Pour pouvoir accéder de manière simple à toutes les ressources qu'offre le système, sans pour cela en rester dépendant à un niveau plus élevé, il est nécessaire de cacher ces différents services dans des primitives compatibles avec le langage utilisé, ce qui permet également une récupération des erreurs plus aisée.
En ce qui concerne la gestion des collections, l'implémentation de cette dernière devient dès lors, un peu moins dépendante du contexte dans lequel elle se situe.
Le développement de cette gestion est justifiée par le fait que le pascal est un moyen d'expression très pauvre au niveau de la gestion des fichiers.

Ces primitives se trouvent dans les unités suivantes : COLLECT,BFIL, XFIL, HFIL.

4.3.1.3. Ecran

A ma connaissance, il n'existe pas de gérant d'écran dédié à l'environnement turbo pascal qui réponde aux exigences suivantes :

- Utilisation d'un environnement standard valable à la fois pour un certain nombre de cartes graphiques mais aussi pour accéder à l'écran physique ainsi qu'à un écran virtuel (zone mémoire) de manière transparente .

- Intégration complète des primitives d'entrées-sorties classiques (read(ln) & write(ln)) pouvant être utilisées dans des contextes différents.

- Intégration complète d'un ensemble de services identiques, quel que soit le mode utilisé (graphique ou texte) : il suffit de voir les différences qu'il y a entre les unités GRAPH et CRT du turbo pascal pour se rendre compte que c'est loin d'être le cas.

- Offrir un moyen d'accès homogène à des objets hétérogènes qui présentent pourtant une certaine unité logique (écran physique, fenêtre, écran virtuel,....)

- ...

C'est pour rencontrer ces exigences qu'il m'a semblé nécessaire de développer un gérant d'écran s'appuyant sur un certain nombre de services identiques, que ce soit en mode graphique ou en mode texte, complétés par quelques primitives particulières au mode considéré.

Ces primitives se trouvent dans les unités suivantes : SYGR, MENU, HARD_CGA, MULTEX, TXDEV.

4.3.1.4. Math

Afin d'être totalement indépendant de la gestion des calculs sur les nombres réels (précision, avec ou sans coprocesseur numérique,...), il suffit d'enfermer dans un

petit module toutes les opérations de base que l'on désire utiliser.
De plus, le turbo pascal n'offrant aucun mécanisme de récupération des erreurs pouvant se produire pendant les calculs, j'ai ajouté quelques primitives permettant de gérer ces erreurs pour autant que l'environnement n'ait pas été trop dégradé par la survenance de l'erreur.

Ces primitives se trouvent dans l'unité CALCUL.

4.3.1.5. Utilitaires

Enfin, voulant disposer d'un certain nombre d'extensions qui ne sont pas offertes par la librairie du compilateur, plutôt que d'écrire des lambeaux de code épars, il me semble plus rentable de développer quelques primitives et de les regrouper dans des unités de manière à pouvoir les réutiliser dans d'autres contextes.
Ces petites primitives permettent de gérer des adresses, des strings de caractères et de mots ainsi que des buffers lors de l'interprétation d'un texte fourni par l'utilisateur.

Elles se trouvent dans les unités suivantes : MIX, XSET, X(p)STR, STRW.

4.3.2. L'interface utilisateur

Du chapitre trois, il ressort que l'interface utilisateur repose sur la manipulation parallèle de deux objets, la 'représentation graphique' du montage et sa 'représentation externe'.

Pour alléger le traitement des modifications de ces objets, les structures de données correspondantes seront implémentées dans deux unités qui offriront un certain nombre de primitives permettant de les manipuler.
Cependant, pour garder une plus grande liberté d'action, il est utile de garder accessible la représentation "physique" de ces objets (la structure de données).

La figure 4.1 donne l'articulation des différentes unités (*units* du turbo 4) destinées à implémenter l'interface

utilisateur. Celles-ci font également appel aux services des unités de gestion d'écran et de menus.

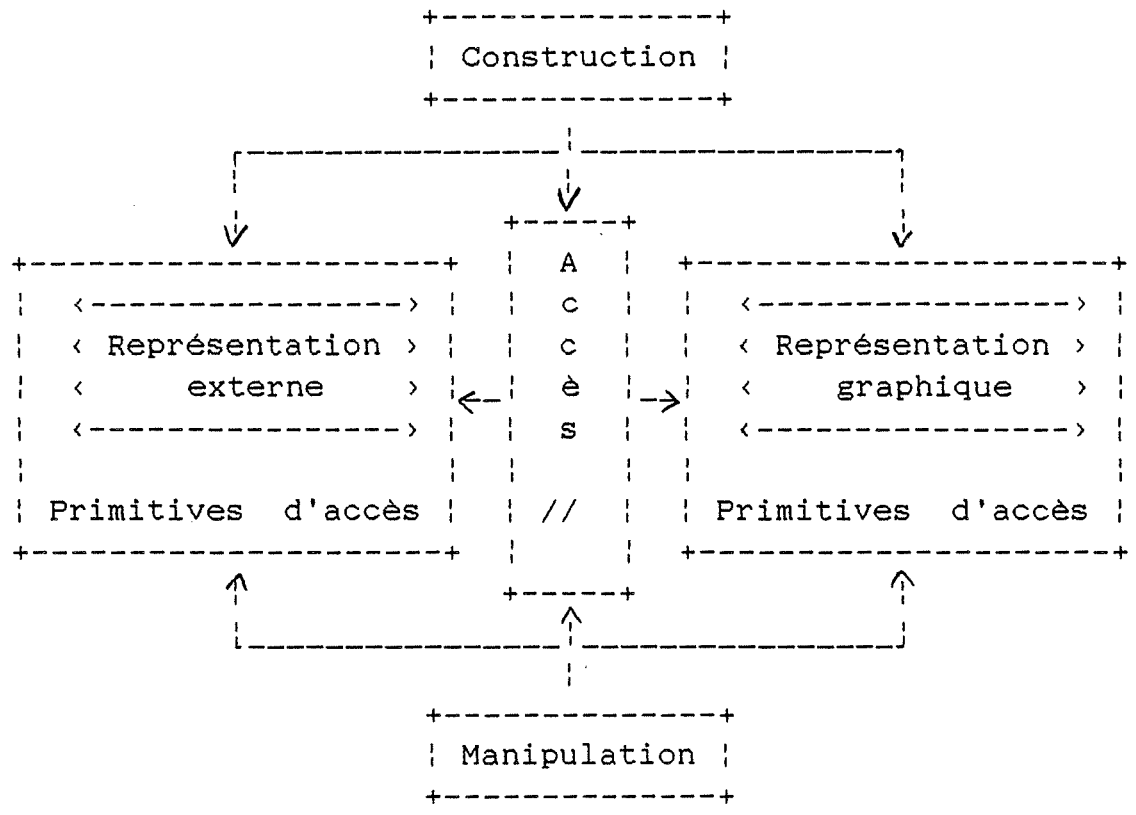


Figure 4.1

On construira le plan en accédant à l'un des objets disponibles dans SYGR. Etant donné que leur utilisation est la même quelle que soit leur nature, le choix du type d'objet à utiliser est entièrement indépendant du développement de ce module-ci.
La représentation externe, quant à elle, prendra la forme d'un tableau à deux dimensions à bornes dynamiques (cfr N(i,j) de la figure 3.3). Tandis que l'ensemble des composants sera implémenté sous la forme d'une pile d'objets hétérogènes (une zone de stockage et une pile de références vers cette zone).

Voyons quelques particularités de l'implémentation de cette structure de données.

Il est intéressant, comme cela est suggéré dans le chapitre trois, de limiter à huit le nombre de branches reliées à un noeud. Cela permet, en effet, de représenter toute l'information contenue dans un noeud par un ensemble de bytes dont les bits témoignent de la présence ou non d'une branche pour chacune des huit directions possibles. On prendra donc un byte pour chacun des types d'informations suivant : les directions possibles (cfr les noeuds en bordure de support), les directions occupées par un composant, les directions occupées par un fil de cablage, les directions interdites (cfr le croisement de composants) et enfin le résumé de toutes ces occupations (accumulation de tous les bytes précédents par un 'or'). Reste alors à prévoir un tableau de références correspondant aux huit directions dans lesquelles un composant peut se trouver.

Le fait de ne plus considérer les fils de cablage comme des composants permet de limiter le nombre total de composants à une quantité raisonnable : 255. Donc leur référence ne prendra pas plus d'un byte non plus.

D'autre part, si on considère une 'cellule' comme quatre noeuds formant les sommets d'un rectangle, les dimensions du tableau de noeuds seront telles qu'il ne pourra représenter qu'environ 320 'cellules'. Cette limite est dictée par le fait qu'un écran virtuel ne peut dépasser 64k (comme tout objet du turbo pascal) et qu'une 'cellule' doit comporter environ 40 points (horizontalement et verticalement) pour que l'on puisse encore distinguer son contenu (une 'cellule' = 40*40 points = 40 * (5 bytes) = 200 bytes et 64k / 200 = 327).
Un second facteur vient encore limiter les possibilités de configurer ces 'cellules' : les lignes d'un écran virtuel ne pourront comporter que 2040 points (255 bytes). On ne pourra donc placer que 51 cellules horizontalement (52 noeuds).
En dehors de ces deux contraintes (327 'cellules' réparties en lignes de maximum 51 'cellules'), il sera possible de définir les dimensions du tableau de noeuds en fonction de la physionomie des circuits à étudier.

Si l'on se réfère à la découpe en modules logiques du chapitre trois, on remarque que la figure 4.1 comporte une unité supplémentaire (accès parallèle).
Elle permet de répercuter de manière uniforme une action ayant des implications simultanées sur le plan du circuit et sur sa représentation interne : ce qui allège la tâche des unités de construction et de manipulation.

4.3.3. Le simulateur

A partir des données du chapitre trois, il ressort que le simulateur s'appuie sur une structure de données (la représentation interne du circuit) dont l'évolution est le reflet du comportement d'un circuit électrique alimenté par une source unique.

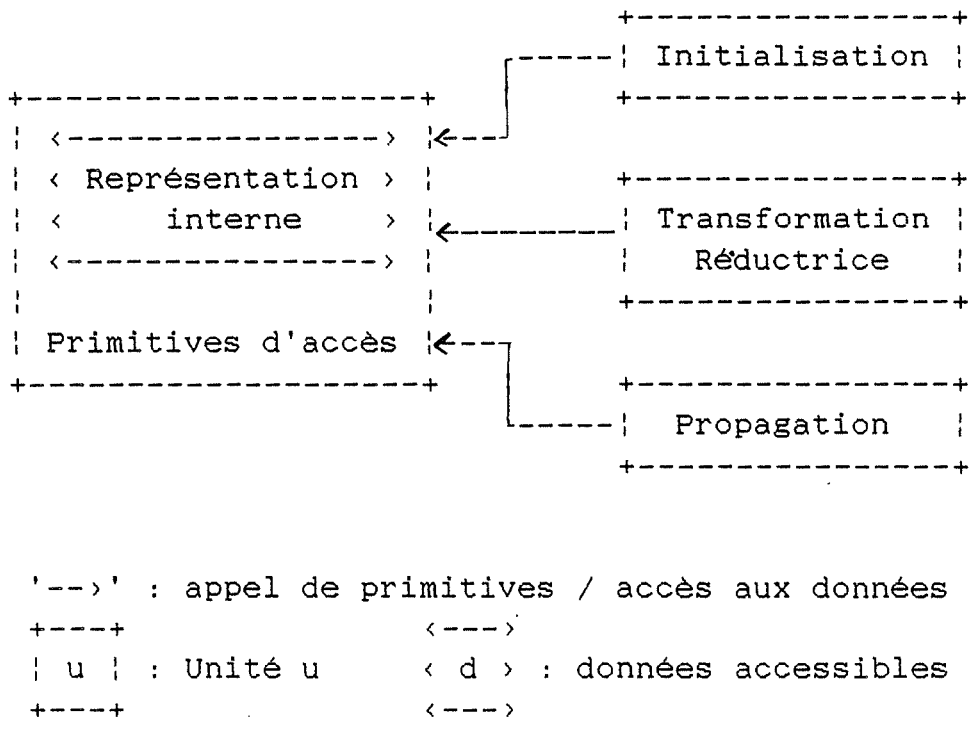


Figure 4.2

On peut distinguer trois phases dans cette évolution : la construction de la représentation interne, la réduction à un composant équivalent et enfin l'éclatement de ce résultat.

Ces trois phases manipulent la même structure de données, ceci donne la découpe présentée à la figure 4.2

La représentation interne du circuit découle logiquement de la structure développée au chapitre trois. Le lien privilégié qui existe avec la représentation externe est constitué par la référence du composant dans cette dernière (un byte ici).
Toute la communication est basée sur cet identifiant.

La taille des composants de la structure développée au chapitre trois sera limitée par le fait que le nombre total de composants ne dépassera jamais 255. Nous réduirons encore ce nombre ici, en disant que le nombre total de branches générées par la réduction ne pourra pas non plus dépasser cette limite. Ceci permet de récupérer les primitives de manipulation de string du turbo 4 pour gérer la suite des références des branches actives. Si l'on garde cette limite pour le nombre de noeuds, on peut appliquer le même procédé pour gérer la suite des références des noeuds actifs.

Les deux ensembles (branches, noeuds) étant des ensembles d'éléments de taille fixe, on peut les implémenter sous la forme de tableaux à bornes dynamiques (même principe que pour les strings : nombre maximum d'éléments & nombre courant d'éléments).

En ce qui concerne les deux derniers composants (étoiles, transformations), leur implémentation se fera sous la forme d'une pile d'objets hétérogènes (cfr point précédent). Néanmoins, pour les étoiles, il est inutile de prévoir la pile de références puisque celles-ci font partie de la représentation des noeuds.

Lors de la phase d'initialisation, le problème majeur réside dans l'identification unique des branches et des noeuds dans les représentation externe et interne du circuit.
Il faut de plus, que les deux modules d'accès aux structures de données soient indépendants.
Il suffit de créer un module intermédiaire qui connaisse les deux représentations et de définir un protocole de

communication entre les deux représentations.
Celui-ci peut être extrêmement simple si l'on reprend l'idée du chapitre trois qui se basait sur le l'ordre dans lequel les branches et les noeuds sont générés dans la représentation interne.
Il suffit au module intermédiaire de retenir la correspondance qui existe entre les identifiants des mêmes objets dans les deux représentations.

Il faut également qu'il puisse fournir les renseignements nécessaires au processus d'initialisation, à savoir :

- Fournir la suite des branches.
- Définir l'alimentation.
- Pour un noeud particulier, compléter la liste des branches qui y sont reliées.
- Pour une branche particulière, connaissant une de ses extrémités, fournir le nombre de branches qui partagent la seconde borne à laquelle elle est reliée.

La phase de réduction renvoie l'information liée au composant équivalent (impédance, courant, tension).

Le module implémentant la phase de propagation permettra de déterminer, à la fin de cette phase, le comportement de chaque branche (courant, différence de potentiel) ainsi que le sens du courant (à partir de deux identifiants d'étoiles (cfr initialisation)).

Comme on peut le constater, les communications avec l'extérieur du module sont extrêmement limitées. Il faut toutefois y ajouter toute la gestion des incidents (demande d'information, 'singularité', échec de la stratégie de résolution) ainsi que le mécanisme de reprise éventuelle.

4.3.4. Les composants du module coordinateur

Dans le chapitre trois, le module coordinateur comporte un certain nombre de composants en plus du programme principal. Ceux-ci peuvent être directement implémentés sous la forme d'unités dont le rôle est d'offrir un ensemble de services permettant de faire le lien entre le simulateur et

l'interface utilisateur (superposition, diagnostic).
Ceci donne donc une découpe du coordinateur en trois unités : le programme principal (SILEX), l'unité de superposition (SUPER) et l'unité de diagnostic (MONITOR).

Le rôle du programme principal est d'orchestrer le fonctionnement de toutes les unités qui sont utilisées dans la réalisation de ce didacticiel. C'est à lui qu'incombe, par exemple, le rôle de communiquer avec l'utilisateur dans tous les cas où il s'agit d'informations qui ne sont plus liées au fonctionnement interne d'une des unités présentées ci-dessus.
En temps que "chef d'orchestre", il dispose principalement des "instruments" suivants :

- Des outils d'entrées-sorties (écran, clavier, 'collections', menus,...)
- Un "éditeur" de circuit (construction)
- Une sonde d'exploration de circuit (manipulation)
- Un transformateur (unité de superposition)
- Un réducteur de circuit
- Un interpréteur de circuit réduit (unité de propagation)
- Un outil de diagnostics

4.3.5. Traitement des incidents

Pour traiter les différents incidents qui peuvent se produire, nous reprendrons les mêmes principes que ceux qui président au traitement des erreurs dans l'environnement du turbo pascal.

Lorsqu'une routine du programme détectera un événement (erreur, 'singularité', break,...), son rôle se réduira à garnir une variable avec un code d'erreur, initialiser un pointeur avec l'adresse depuis laquelle l'appel s'est effectué et enfin appeler une routine de traitement d'exception dont l'adresse est contenue dans une variable de type pointeur.
Ce genre de démarche permet de centraliser la gestion de la reprise de l'erreur tout en permettant d'ignorer certaines exceptions en fonction du contexte d'utilisation.
De plus, cette procédure générale de traitement des erreurs

est la même que celle qui est utilisée par n'importe quel programme généré par le compilateur du turbo pascal lorsque celui-ci se termine.
Enfin, étant donné la manière dont la librairie "*system*" du turbo pascal gère l'interruption générée par le système d'exploitation lorsque survient une erreur grave (int 24h : critical error handler du MS-DOS), il devient également possible de récupérer ces erreurs par le même moyen.

L'approche proposée est donc parfaitement adaptée à l'environnement turbo pascal et de plus, elle permet de gérer de manière homogène n'importe quel type d'exception.

CONCLUSION

Le moment est venu d'évaluer dans quelle mesure notre objectif de départ est atteint.
Si l'on se réfère à l'introduction, le but était de voir si l'ordinateur pouvait être d'une quelconque utilité au niveau d'un laboratoire d'électricité.

Le chapitre un a montré le principal avantage que l'on pouvait tirer de l'utilisation de l'informatique pour simuler un tel laboratoire : l'amalgame d'objets inanimés qui constituent les éléments de ce dernier se mettent à "vivre" et donnent ainsi une toute autre dimension à son utilisation pédagogique.
Malheureusement, l'informatique ne permet pas de simuler un laboratoire d'électricité aux possibilités infinies. Contraint à la spécialisation, il a fallu en limiter les applications.

Partant de là, nous avons cerné la physionomie des circuits à étudier tant du point de vue de ses éléments constitutifs, que de leur comportement.

Le chapitre deux a abouti à la définition d'une stratégie de résolution permettant de simuler le comportement des types de circuits envisagés.
A ce moment là, tous les éléments nécessaires à la construction du didacticiel étaient réunis.
C'est ici qu'intervient un des termes de l'objectif : il s'agit de construire une boîte à outils efficaces et réutilisables.
Il faut donc que ceux-ci soient d'un usage assez général pour qu'ils puissent être utiles dans d'autres contextes.

Cette idée a guidé toute la découpe en modules du didacticiel qui fût l'objet du chapitre trois.
Il ne restait plus qu'à poser les derniers jalons nécessaires à la réalisation concrète d'un autre terme de l'objectif : la boîte à outils doit être utilisable directement....

En ce qui concerne le programme exécutable, on peut dire que l'objectif n'est pas entièrement atteint. Il l'aurait sans doute mieux été si je n'avais pas choisi l'optique boîte à outils. Cependant, cette optique est primordiale à mes yeux, surtout en regard d'un programme dont on dit toujours qu'il peut être amélioré.
Par qui ? Déjà pour son auteur c'est souvent difficile, alors pour quelqu'un d'autre... Dans le cadre d'une boîte à outils, si ces derniers ne sont pas trop mauvais, c'est différent. On commence par les utiliser parce qu'ils permettent de résoudre à moitié un problème. Ensuite, le fait de s'en servir peut devenir une invitation à les compléter.... ou à les remplacer par d'autres qui seraient meilleurs.

Cependant, si l'on regarde la démarche que j'ai voulu la plus claire possible et ce, pour le plus grand nombre de personnes, on peut y déceler, je crois, l'amorce d'une réflexion qui pourrait être poussée dans plusieurs directions.

PROLONGEMENTS

Dans un premier temps, on pourrait très bien envisager d'enrichir la boîte à outils par des éléments qui permettraient de les utiliser pour d'autres applications que le principe de superposition (théorème de Thevenin).

Ensuite, il faudrait trouver un moyen de simuler les transitoires d'un circuit électrique, développer des outils d'analyse de sensibilité,...

A un niveau plus général, on pourrait aussi concevoir des outils destinés à simuler d'autres phénomènes qui présentent les mêmes particularités qu'un laboratoire d'électricité : une représentation graphique qui cache un certain nombre de principes théoriques.
Je pense particulièrement à un cours de géométrie plane où les composants électriques seraient ici des formes géométriques, et les tranformations de réduction, des propriétés à partir desquelles il faut construire une démonstration....
Ceci n'est évidemment plus compatible avec les contraintes que l'on s'est fixé au départ, mais un certain nombre de composants ayant servis à la résolution de ce problème peuvent être récupérés. (Au niveau des idées, on pourrait s'inspirer des mécanismes de gestion des structures de données. Au niveau de l'implémentation, on peut récupérer un bon nombre d'outils.).

Enfin, lors de l'étude du simulateur, je me suis rendu compte qu'il aurait été utile de pouvoir disposer d'un certain nombre d'outils de représentation et de gestion de données qui soient plus directement en rapport avec les concepts manipulés. En effet, si dans le cadre de ce mémoire, les fragments de connaissance, directement inclus dans la structure de données qui les représente, étaient peu complexes, on imagine sans peine que pour d'autres matières, il en va tout autrement.
Or, il me semble possible, même à l'aide de langages de programmation classiques, de pouvoir construire un certain nombre de techniques de représentation de ces fragments de connaissance qui allégerait considérablement certaines phases du développement de didacticiels.

QUELLES LECONS EN TIRER ?

Dans le cadre d'une licence en informatique, on peut dire que le développement d'un didacticiel est riche d'enseignement. En effet, le contexte dans lequel ce développement doit avoir lieu est, à mon avis, une préfiguration du devenir de l'informatique.

Dans l'introduction, j'ai parlé des quatre acteurs en présence (concepteur, didacticiel, enseignant, étudiant) qui sont la source d'une grande différence entre le développement de didacticiel et de logiciel classique (où les trois acteurs sont alors le concepteur, le logiciel et l'utilisateur).
L'enseignant est ici un utilisateur privilégié qui doit pouvoir modifier le didacticiel pour l'adapter à l'objet précis de son cours.
Or, si l'on regarde ce qui se passe dans le domaine de l'automatisation, on s'aperçoit que pour rentabiliser au maximum les investissements requis par une machine à commande numérique, il faut que l'utilisateur passe du stade "presse-bouton" à celui d'une maîtrise plus grande de l'outil mis à sa disposition (programmation).

Ce phénomène, à mon avis, se rencontrera tôt ou tard dans l'informatique "traditionnelle". Pour que les investissements nécessités par celle-ci s'avèrent plus rentables, il faudra une plus grande implication de l'utilisateur. On se retrouvera donc dans la même situation que pour l'enseignement assisté par ordinateur : quatre acteurs au lieu de trois. Et parmi ceux-ci, il y aura deux utilisateurs dont un, au moins, devra être à même de modifier le logiciel pour l'adapter à ses besoins propres....

PISTES de REFLEXION

Sans avoir la prétention d'être exhaustif, quelques thèmes de réflexions me semblent dignes d'être approfondis dans le cadre de l'EAO.

Le premier a trait au problème du rôle de l'enseignant par rapport au didacticiel.
En effet, permettre à un utilisateur (l'enseignant) d'agir sur le fonctionnement d'un logiciel de manière à modifier légèrement la finalité est une véritable gageure. Un concepteur de logiciel a déjà toutes les peines du monde à savoir si le programme qu'il a conçu fonctionne correctement, que dire s'il ne connaît pas à priori la destination exacte de ce dernier ?

En supposant que tout fonctionne correctement, par quel moyen va-t-on permettre à l'enseignant d'adapter un didacticiel ? L'approche du type "langage auteur" est vouée à l'échec, en ce sens qu'il s'agit de fournir un compilateur déguisé pour ne pas effrayer un non-informaticien. Autant appeler les choses par leur nom et dire alors que les enseignants doivent devenir des informaticiens sachant au moins utiliser un langage conventionnel comme le pascal...
Cela me semble impraticable à long terme. En effet, ce type d'attitude ne fait que déplacer le problème en disant aux enseignants de se débrouiller. Pourtant, il s'agit d'un défi lancé aux informaticiens, pas aux enseignants.
Il ne reste donc plus qu'à trouver un moyen ne requérant qu'un minimum de connaissances informatiques, par lequel un enseignant puisse communiquer au didacticiel la manière dont il doit se comporter.

Enfin, du côté de la conception de didacticiel, il faudrait créer un environnement de développement. Celui-ci devrait permettre la mise au point de nouveaux didacticiels à partir d'embryons de solutions déjà utilisés, limitant ainsi les efforts de développement nécessités par chaque nouvelle application. En diminuant le temps nécessaire au développement, on apporte une partie de la réponse au problème du didacticiel "sur mesure". Si l'enseignant n'a plus qu'à le modifier très légèrement pour l'adapter à l'objet de son cours, il devient alors possible d'envisager cette phase comme une initialisation de configuration de programme.
C'est donc vers la mise au point d'un environnement de développement qu'il faudrait se tourner dans un premier temps.

BIBLIOGRAPHIE

F.A. BENSON & D. HARRISON
Electric-Circuit Theory
Edward ARNOLD, Londres, (3° ed. 1975).

Donald A. CALAHAN
Computer aided Network design
McGRAW HILL, New-York, (1972 : ed. 1968 révisée)

Joseph A. EDMINISTER
Théorie et application des circuits électriques
Série SCHAUM (Groupe McGRAW HILL)

Wai-Kai CHEN
Applied graph theory
NORTH-HOLLAND Publishing Company, Amsterdam, (1971)

TABLE des ANNEXES

Silex

Construction

Simulation

TABLE des ANNEXES

TABLE des ANNEXES

Hardware vidéo

Extensions de System

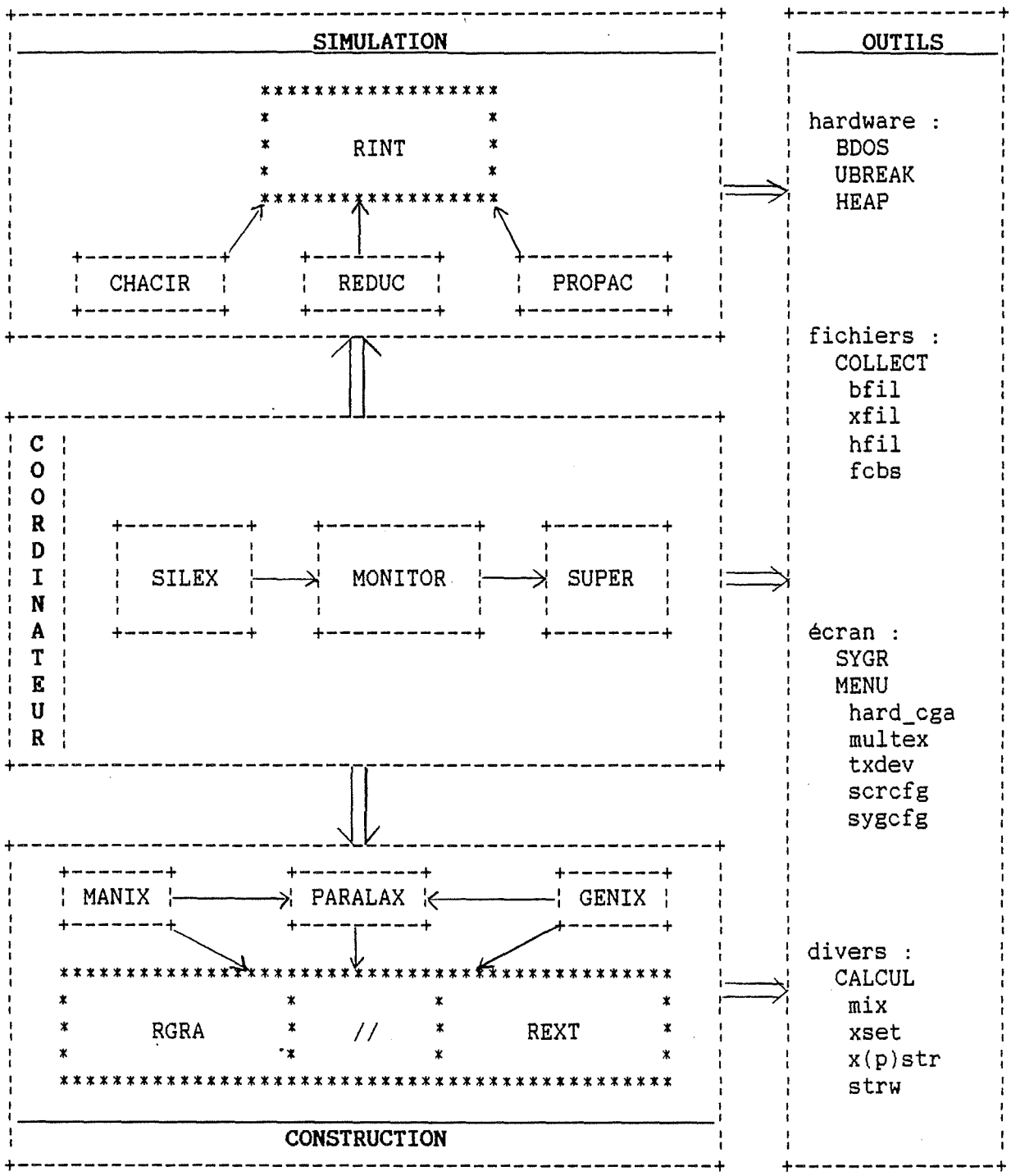
Gestion de fichiers

TABLE des ANNEXES

Le bios

Utilitaires

ANNEXE I. Module principal



`+---+ ¦ S ¦ +---+`	: Ensemble de services
`'==›'`	: utilisation logique
`***** * d * *****`	: Accès à une structure de données
`'--›'`	: appel de primitives

Structure générale de Silex

1.1. Présentation générale de SILEX

Je ne détaillerai pas ici le code du module principal. Je me limiterai uniquement à détailler les aspects globaux de l'utilisation des outils présentés ci-dessus.

Ces aspects peuvent être regroupés en deux classes :
- La paramétrisation des différentes unités
- Les possibilités de récupération d'erreurs (ainsi que les codes correspondants).

1.2. Paramétrisation des outils

On trouve ci-dessous la liste des possibilités de paramétrisation statique des différents outils. Il s'agit d'une part de la taille des différents objets manipulés et d'autre part du mode de gestion d'occurence d'événements (erreur, break).

SYGR
- cfr ScrCfg et SygCfg(type de carte graphique, nombre d'objets)
- router (@ routine de traitement des erreurs)
- Ubreak.on_break (@ routine de traitement du break : write(ln)/read(ln))

MENU
- cfr SygCfg (nombre d'objets)
- clavier (@ routine d'entrée des caractères)

(Le break est détecté par l'occurence du caractère 0000H venant du clavier)

COLLECT
- rhndl (@ routine de traitement des erreurs)
- atcre (valeur de l'attribut -› création de fichiers)
- path (localisation de TOUTES les collections ouvertes)

REXT-RGRA-PARALAX-MANIX-GENIX
- rgra.modgra (apparence graphique des composants)
- paralax.canac (canal d'accès -› représentation graphique)
- genix.caplan (type du canal d'accès paralax.canac)
- genix.rplan (taille des éléments de la représentation externe)

RINT-CHACIR-REDUC-PROPAC
- Rint.simur (@ routine de traitement des erreurs)
- Rint.def_rin (taille des éléments de la représentation interne)

(Le break est détecté par ubreak.break)

1.3. Récupération des erreurs

Avant de passer à la récupération des erreurs proprement dites, il est utile de signaler le moyen pour qu'un break utilisateur n'interfère pas avec le déroulement du programme. Il suffit de l'interdire par la primitive ubreak.breaksw. Le seul endroit où l'interférence aura toujours lieu, c'est dans l'unité de gestion menus (sortie de menu).

Ayant mentionné dans le point précédent toutes les possibilités de définir

une routine de traitement de l'occurence d'une erreur, je vais maintenant passer en revue tous les codes d'erreurs possibles ainsi que leur signification. Il est bien entendu que contrairement à ce qui se passe pour les erreurs d'entrée-sortie (cfr ioresult), on peut ignorer totalement l'occurence d'une erreur (deux erreurs successives n'entrainent pas la terminaison du programme).
Seul le code relatif à la dernière erreur est disponible.
Dans certains cas (cfr simulation), ce code d'erreur est complété par une série d'informations.
Attention, le fait de se servir du code d'erreur pour recommencer la même opération que celle qui a entrainé l'erreur peut se traduire par le bouclage indéfini du programme.
En laissant plus de possibilités de contrôle sur les outils, on accroît également les responsabilités de celui qui les utilise !

Les différentes variables pouvant contenir un code d'erreur sont :
- Sygr.coder
- Collect.coler
- Rext.rextr
- Rint.rintr
- super.supr
- calcul.mater

Le fait qu'il en existe plusieurs peut sembler inutile à première vue, mais c'est la seule façon de faire si l'on veut préserver la modularité des outils qui permet de prévoir des politiques de récupération différentes en fonction de l'origine des erreurs.
Néanmoins, pour ne pas se perdre dans une multitude de codes différents pour un même type d'erreur, tous les codes seront les mêmes, quel que soit le module qui les génère.
Lorsque c'est possible, je reprendrai les codes correspondant aux runtime errors du turbo 4. On ne retrouvera ci-dessous que les codes différents de ceux du turbo 4 ou bien ceux qui nécessitent un commentaire supplémentaire. Les autres codes du turbo 4 conservent leur signification.

0 Pas d'erreur

1 Bug dans le programme
Elle correspond souvent à une mauvaise utilisation d'une interruption du dos.

2-18 Erreurs du dos
Ces erreurs correspondent à celles qui sont générées par le dos, elles sont reprises également par le turbo 4.

99 Break de l'utilisateur
Lorsqu'un break de l'utilisateur a interrompu le déroulement d'une primitive.

100 Erreur de lecture disque
Si l'on essaie de lire après la fin d'une collection.

101 Erreur d'écriture disque
Si l'on essaie d'écrire sur une disque plein.

106 Mauvais format numérique
Erreur de conversion texte‹-›nombre (calcul.???)

107 Trop d'objets définis
Lorsque l'on tente de définir un nouvel objet alors qu'il n'y a plus de descripteur libre (cfr Sygr, collect,...).

108 Mauvaise référence d'objet
Si l'on essaie d'accéder à un objet non défini (cfr Sygr, collect...).

109 Incohérences dans les paramètres
Lorsque les paramètres passés à une primitive comportent des incohérences.

110 Objet trop grand
Lorsque l'on essaie de définir un objet dont la taille est supérieure aux limites fixées (›64k par exemple).

111 Mauvais stockage
Lorsqu'un module n'a pû sauver ou récupérer un objet grâce à la primitive dont on lui a donné la référence (s'il s'agit d'une primitive de collect, la nature de l'erreur peut alors être déterminée à partir de coler).

112 Erreur de modification d'un objet
Lorsqu'une modification demandée pour un objet est aberrante (par exemple demander de mettre un titre à une fenêtre sans bord,...).

113 Objet inaccessible
Lorsque l'on tente d'accéder à un objet défini mais non accessible (par exemple à une fenêtre partiellement masquée,...).

114 Localisation interdite
Erreur générée lors de la définition de la localisation d'un objet incompatible avec son environnement (mauvaise position d'une fenêtre par exemple, masque d'accès trop restrictif, définition d'un bord réduisant à rien la zone utile,....)

115 Erreur de calcul
Signale qu'une erreur a été récupérée par le module calcul.

116 Objet inconnu
Lorsqu'un module ne reconnait pas la structure de l'objet qu'on lui propose (rext.assign_rec par exemple).

117 Erreur de construction
Signale l'adjonction erronée d'un élément à une structure de données lorsque cette dernière peut être construite par étapes.

118 Action interdite
Lorsque l'on utilise une primitive alors qu'elle ne peut plus l'être dans le contexte courant (cfr chacir.cleb par exemple).

119 Overflow de la structure de données
Lorsque l'évolution dynamique d'une structure de données est empêchée par sa trop petite taille (par exemple lorsqu'une borne dynamique devrait dépasser la valeur maximale qu'elle peut prendre).

120 Ambiguïté de la structure
Lorsqu'une indétermination survient au cours d'un processus.

121 Court-circuit
Lorsqu'une structure de données comporte un cycle de références qui ne devrait pas s'y trouver.

122 Dépassement de limites
Lorsque l'on tente de dépasser les limites d'une structure (de données) lors de l'utilisation d'une primitive d'accès (mauvaise position du curseur, dépassement de borne d'un tableau,...)

ANNEXE II. Module de surveillance

2.1. Présentation générale de MONITOR

Le rôle de ce module est d'implémenter le mécanisme de communication des diagnostics posés par le didacticiel.
Il s'agit donc "d'enrober" un certain nombre de primitives offertes par les différents outils de manière à récupérer un certain nombre d'erreurs.
Bien entendu, des erreurs sont impossibles à récupérer (par exemple manque de mémoire vive), dans ce cas, c'est au module principal de les gérer.
Ce module sera donc chargé de la "surveillance" de la simulation du circuit dont une représentation externe est définie et pour laquelle une structure de données de la représentation interne existe.
Son interface sera donc extrêmement réduite.

2.2. Présentation de l'interface

```
unit monitor;
INTERFACE
uses rext,manix,          {préparation du dialogue -› utilisateur}
     super,               {préparation de la simulation (+ superposition)}
     rint,reduc,propac;   {conduite de la simulation}

type nlog   = array[1..mxCmp] of Nij;
     Pnlog  = ^nlog;

var  causer : cause;      {cause probable de l'erreur}
     messer : ^string;    {‹›nil => commentaire -› erreur}
     eter   : Pnlog;      {noeud logique -› erreur}
     comer  : PVER;       {branche(s) -› erreur}

function  SIMULER : boolean;
{    Primitive de simulation d'un circuit renvoyant la valeur TRUE en cas
     de succès. Sinon la valeur false est renvoyée et les variables causer,
     messer, eter et comer sont initialisées.
}

IMPLEMENTATION {monitor.pas}
```

ANNEXE III. Principe de superposition

3.1. Présentation générale de SUPER

Le rôle de ce module est d'implémenter le principe de superposition en utilisant les outils de manipulation de la représentation externe et ceux de la représentation interne. Pour préserver la possibilité d'utiliser ce module dans divers contextes, il sera chargé uniquement de l'initialisation de la simulation et de la répercussion des résultats de celle-ci sur la représentation externe. Pour éviter que de nombreux calculs ne soient refaits plusieurs fois lorsqu'un circuit comporte plusieurs sources travaillant toutes à la même fréquence, ce module offre également la possibilité de mémoriser la valeur des impédances des différents composants.
On peut donc dire que ce module offrira trois types de services :

- Calcul de l'impédance des composants.
- Génération de la représentation interne d'un circuit pour lequel on veut connaître l'influence d'une source particulière.
- Superposition des résultats obtenus pour chaque source.

Lorsqu'une erreur survient, un code d'erreur est généré et la référence des éléments litigieux de la représentation externe est définie. Aucun mécanisme de reprise n'est prévu (En effet, les anomalies graves (erreurs de cablage) seront détectées pendant la phase d'initialisation, tandis que les autres nécessiteront un redémarrage complet de la simulation).
Ceci définit le comportement par défaut de toutes les primitives de super en cas d'erreur.

3.2. Présentation de l'interface

```
unit super;
INTERFACE
uses heap,calcul,mix,   {les outils de bas niveau}
     rext,rint,         {les deux représentations}
     chacir;            {initialisation de la représentation interne}

type  PVER = ^VER; {cfr rext}
      idal = Rcomp; {identifiant interne à super -› source}

const supR : byte = 0; {code d'erreur}
var   nodR : Nij;      {noeud -› erreur}
      braR : Rcomp;    {composant -› erreur}

procedure SCANCIR(var d : DREXT; var p : PVER);
{     Primitive d'initialisation du processus d'étude du circuit d.
      Une représentation logique (noeuds logiques) du circuit est construite
      ainsi que la suite p des sources d'alimentation dont l'identifiant
      devient leur position dans cette suite (1..255).
      Les structures de données construites lors d'un appel précédent sont
      détruites par tout nouvel appel de scancir.
      Cette procédure s'assure qu'aucun composant n'est en court-circuit
      (origine et destination identiques).
}
```

```
procedure DELSCAN;
{    Primitive de destruction de toutes les données internes au module
     (permet la récupération de la mémoire).
}

procedure GENINT(a : idal; sauz : boolean);
{    Primitive de génération de la représentation interne du circuit
     alimenté par a. (annihilation de l'effet des autres sources).
     Il faut que la représentation interne courante existe déjà (cfr
     chacir.newcir). La valeur des impédances de chaque composant est
     gardée si sauz a la valeur TRUE (mémorisé jusqu'à scancir ou delscan).
}

procedure VAZ(a : idal; b : Rcomp; var z : cmplx);
{    Primitive de calcul de l'impédance z d'un composant b relative à la
     source a. Si le calcul a déjà été effectué (genint), il ne sera pas
     refait : renvoi immédiat du résultat.
}

procedure SIMEX;
{    Primitive de superposition du résultat de la simulation du comporte-
     ment du circuit par rapport à une source (Rint) sur le comportement
     global courant du circuit (Rext). S'il s'agit de la première superpo-
     sition depuis scancir, le résultat de la simulation (courant, tension,
     sens du courant) est simplement répercuté tel quel (sinon c'est une
     somme du contenu courant de Rext avec celui de Rint).
}

IMPLEMENTATION {super.pas}
```

ANNEXE IV. Représentation externe

4.1. Présentation générale de REXT

Comme cela a été présenté dans les chapitres trois et quatre, la représentation externe se compose de deux éléments : le tableau à deux dimensions représentant les noeuds et une pile d'objets hétérogènes.
Le rôle de ce module est d'offrir des primitives de gestion de cette représentation. De plus, il doit permettre une gestion simple du stockage de cette représentation dans un fichier.
C'est pourquoi, il sera basé sur un descripteur de la représentation externe.

4.1.1. Stucture de données.

La structure de données doit permettre de représenter 320 cellules rangées en lignes ne comportant pas plus de 51 cellules.
Il s'agit donc d'un tableau à bornes dynamiques dont les dimensions logiques maximales sont de 51 lignes de 51 cellules. J'ai opté pour une matrice carrée plutôt que de permettre 320 lignes d'une cellule qui sont des dimensions peu réalistes.
A partir de ces dimensions logiques, on peut déduire que le tableau ne pourra comporter que 52 lignes de 52 noeuds.
Les trois paramètres décrivant cette structure sont donc son adresse, le nombre de lignes et le nombre de noeuds par ligne.

En ce qui concerne les composants, nous en avons limité le nombre à 255. Dans les chapitres trois et quatre, on a dit que ces composants étaient hétérogènes, donc de tailles différentes. Néanmoins, en prenant certaines conventions, on peut faire en sorte qu'ils soient tous de la même taille. C'est l'option que j'ai choisie.
La structure de données aura donc la forme d'un tableau à une dimension de 255 éléments maximum. Il sera accompagné d'un tableau de références permettant de repérer rapidement les places libres du tableau (un string).
Les trois paramètres décrivant cette structure sont donc les adresses des deux tableaux (tableau de composants et string descripteur) ainsi que leur taille physique (nombre d'éléments).

4.1.2. Primitives de gestion

A partir des paramètres décrits ci-dessus, le module doit être en mesure de générer la structure de données correspondante représentant un circuit vide (aucun élément).
Toutes les autres primitives sont basées sur la notion de représentation courante. Lorsque celle-ci est indéfinie, elles échouent toutes.
Il doit ensuite offrir des primitives d'accès à la structure de données de manière à rendre transparent le fait que l'on manipule des tableaux à bornes dynamiques.
Enfin, un certain nombre d'actions logiques (liées à la signification de la représentation externe) sera offert sous la forme de primitives.
Etant donné que le descripteur est visible de l'extérieur, la gestion du stockage de la représentation sera limitée à la fourniture d'adresses physiques d'agrégats d'éléments dont on mentionnera également la taille.

4.2. Présentation de l'interface

```
unit rext;
INTERFACE
uses heap,   {allocation de la structure de données}
     calcul, {valeur des composants,....}
     mix;    {points cardinaux}

const MxDir = {0..}7; {8 directions possibles -› noeud}
      Mxdim = 52;
      MxCmp = 255;

      rextR : byte = 0; {code d'erreur}

type  dim   = 0..MxDim; {-›dimensions logiques max du tableau}
      Nelem = 0..MxCmp; {-›dimension logique du tableau de références}
      Rcomp = 0..MxCmp; {(byte) type de la référence d'un composant}

      Nij   = record li,co : dim; end; {réf d'un noeud}

      tycmp = (RESI, {résistance       }
               SELF, {self             }
               CAPA, {capacité         }
               SOUC, {source de courant}
               SOUT  {source de tension}
                   );{types de composants}

      DREXT = record Rnod : pointer; {@ du tableau}
                     Cnod : dim;     {nbr de noeuds par ligne}
                     Lnod : dim;     {nbr de lignes de noeuds}
                     Rcmp : pointer; {@ du tableau de composants}
                     Rref : pointer; {@ du string de références}
                     MxEl : Nelem;   {nombre max de composants}
                     end; {descripteur de la structure}
      Pdrxt = ^drext; {pointeur -› descripteur}

      Noeud = record Visi  {vrai s'il fait partie du circuit}
                           : boolean;
                     Dpos, {direction impossible = 1}
                     Dint, {direction interdite = 1 (croisements)}
                     Docc, {direction occupée = 1 (composant)}
                     Dcab, {direction occupée = 1 (fil)}
                     Dlib  {direction libre = 0}
                           : byte;
                     Cotab {références des composants/direction}
                           : array[0..7] of Rcomp;
                     end;{structure d'un noeud}
      Pnod  = ^Noeud;
```

```
      Compo = record tyc      {type du composant}
                      : tycomp;
                     valeur   {valeur du composant
                                    (source -› fréquence)}
                      : reel;
                     cournt,  {courant}
                     ddp      {différence de potentiel}
                      : cmplx;
                     RN1,     {le courant traverse ce composant }
                     RN2      {dans le sens RN1 --› RN2          }
                      : Nij; {référence des 2 noeuds : N1,N2}
                     end;{structure d'un composant}
      Pcomp = ^Compo;

      LIN   = array[1..MxDim] of Noeud; {ligne de noeuds}
      VEC   = array[1..MxCmp] of Compo; {vecteur de composants}
      VER   = string[MxCmp];  {vecteur de références de composants}

{PRIMITIVES D'INITIALISATION}
procedure DEF_REC(var d : DREXT);
{     Primitive d'initialisation. d devient la structure de données courante qui est ensuite initialisée.
      En cas d'échec, cette dernière est indéfinie et un code d'erreur est généré (rextR).
      Il faut au moins que les tailles des objets soient définies (Cnod, Lnod, MxEl). Si un des pointeurs correspondant (Rnod, Rcmp, Rref) a la valeur nil, la primitive tente de procéder à l'allocation de la mémoire nécessaire. Tout objet dont la référence est différente de nil est supposé exister et avoir la taille adéquate (Cnod, Lnot, MxEl).
      La structure de données est ensuite initialisée de manière à représenter un circuit vide.
}

procedure SUP_REC;
{     Primitive de destruction de la structure de données courante (restitution de la mémoire, mise à nil des pointeurs) sans modification des paramètres Cnod, Lnod et MxEl.
      Après destruction, la représentation courante est indéfinie.
}

Procedure ASSIGN_REC(var d : DREXT);
{     Primitive permettant de changer globalement de représentation courante. L'éventuelle représentation préexistante est simplement et remplacée par celle-ci sans être modifiée d'aucune façon (pas de restitution de la mémoire,...). Il faut que la nouvelle représentation soit entièrement initialisée.
}

{PRIMITIVES D'ACCES}
function Rnoeud(i,j : dim) : Pnod;
{     Fonction donnant la référence du noeud i,j si elle existe, sinon elle renvoie la valeur nil.
}
```

```
function Rcompo(i : Rcomp) : Pcomp;
{     Fonction donnant la référence du composant i s'il existe, sinon elle
      renvoie la valeur nil. Un composant n'existe effectivement que s'il
      existe logiquement (càd appartenant au circuit).
}

procedure NewComp(var i : Rcomp; var r : Pcomp; N1,N2 : Nij);
{     Primitive de définition d'un nouveau composant. En cas de succès, le
      composant appartient au circuit (sinon aucun changement et r=nil).
      Pour les noeuds N1 et N2, si toutes les contraintes liées à la
      structure du circuit sont respectées, les champs (Visi, Docc, Dlib,
      Cotab) seront affectés de manière à tenir compte de la nouvelle
      connexion.
      De même pour le composant dont la référence i et l'adresse r seront
      fournies (en cas d'échec, r=nil), seuls les champs (RN1 et RN2)
      prendront respectivement les valeurs N1 et N2.
}

procedure SupComp(i : Rcomp);
{     Primitive de destruction d'un composant faisant partie du circuit. Si
      c'est bien le cas, elle effectue l'opération inverse de celle de la
      primitive NewComp.
}

{PRIMITIVES D'INTERROGATION}
procedure LogNoeud(i,j : dim; var gn : word; var sr : VER);
{     Si le noeud i,j fait partie du circuit, alors sr contiendra la suite
      de toutes les références des composants reliés au noeud logique
      considéré. Le noeud logique est constitué du noeud i,j ainsi que de
      tous les noeuds qui y sont connectés par un fil de cablage. gn
      indiquera de combien de noeuds est constitué le noeud logique (au
      moins 1 correspondant au noeud de départ). Si gn est nul, c'est que
      la procédure a échoué (le noeud i,j n'appartient pas au circuit par
      exemple).
}

function alim (var i : Rcomp) : boolean;
{     Primitive de recherche d'une source d'alimentation à partir du compo-
      sant i. Si une source existe parmi les composants d'indices apparte-
      nant à l'intervalle [i,MxCmp], alors la valeur TRUE est renvoyée et i
      contient la référence de la source. En cas d'échec, la valeur FALSE
      est renvoyée et i est indéterminé.
}



{
PRIMITIVES DE COMPACTAGE -> STOCKAGE.
Après compactage des noeuds, il n'y a plus de lien direct avec la représen-
tation graphique du circuit (c'est comme si on déplaçait le plan du circuit
dans le coin supérieur gauche du support graphique).
Comme il y a maintien de la cohérence interne des références, ces opéra-
tions de compactage ne sont pas gratuites (temps !).
}
```

```
function SuiComp : Mxcmp;
{     Primitive de compactage des composants.  Après compactage, le tableau
      des composants ne comporte plus de trous et le nombre total de compo-
      sants est renvoyé.
}

function PlaNod : word;
{     Primitive de compactage des noeuds.  Après compactage,  l'ensemble du
      circuit  occupe un "rectangle" dont le coin supérieur gauche  corres-
      pond  au noeud(1,1).  Le nombre total de noeuds (y compris les noeuds
      intérieurs  ne faisant pas partie du circuit) renvoyé sous  la  forme
      d'un mot dont le byte de poids fort correspond au nombre de lignes de
      noeuds et le byte de poids faible au nombre de noeuds par ligne.
}

function taille_lin (col : dim) : word;
function taille_vec (nc  : Rcmp) : word;
{     Primitives renvoyant la taille d'objets compactés. A savoir la taille
      d'une ligne comportant col noeuds,  et la taille d'un vecteur compor-
      tant nc composants.
}

IMPLEMENTATION {rext.pas}
```

ANNEXE V. Représentation graphique

5.1. Présentation générale de RGRA

Ce module est destiné à fournir un ensemble de primitives permettant de construire le plan d'un circuit.
Comme ce module ne fait aucune autre supposition concernant la nature du canal d'accès, il peut très bien être utilisé pour n'importe quel type d'objets manipulables par Sygr (fenêtre (virtuelle), écran, vue).

On supposera toujours que le canal courant permet d'accéder à la représentation graphique du plan dans des conditions standards (cfr INI_CAN).
On considérera de plus que le coin supérieur gauche de la zone utile du canal d'accès correspond exactement au point supérieur gauche de la représentation d'un noeud faisant partie du circuit.

Une primitive donnera la possibilité de paramétrer le comportement de ce module (cfr interface).
En ce qui concerne les directions que peuvent prendre les composants à partir d'un noeud, on reprendra ici les mêmes conventions que celles utilisées dans le module REXT (cfr constantes des points cardinaux).
Ce module ne mémorise absolument rien à propos de la structure du circuit, sa seule fonction est d'offrir un certain nombre de primitives facilitant la construction graphique du circuit.
Si l'on veut récupérer certaines erreurs, il faut dès lors utiliser le mécanisme de gestion des erreurs du module Sygr.
On procédera de même en ce qui concerne le stockage éventuel de la représentation graphique obtenue.

5.2. Présentation de l'interface

```
unit rgra;
INTERFACE
uses Sygr, {zone utile du canal courant}
     mix;  {points cardinaux}

type  ARCEL  = record licel, {nombre de lignes de cellules}
                      cocel  {nombre de cellules par ligne}
                             : byte;
                      end;
      coins  = (NSG, {noeud supérieur gauche}
                NSD, {  "       "     droit }
                NID, {  "   inférieur droit }
                NIG  {  "       "     gauche}
                   );{les quatre coins d'une cellule}
      corcel = record x1,y1,x2,y2 : corsup end;
      {localisation d'une cellule dans la zone utile du canal courant}

var   plan   : arcel; {taille du plan}
      ccour  : arcel; {cellule courante}
      celoc  : corcel;{localisation de la cellule courante}
```

```
{PRIMITIVES DE DEFINTION}
procedure CEL_COUR(i,j : byte);
{     Primitive de définition de la cellule courante. Si la cellule n'exis-
      te pas (cfr plan), il n'y a pas de changement de la cellule courante
      qui initialement est la cellule (1,1)
}

procedure MORGRA(cpix,lpix,poff,pon,tcmp : byte);
{     Primitive de définition des paramètres de la représentation
      graphique (valeurs par défaut).
      cpix  : dimension horizontale d'une cellule (40 points)
      lpix  : dimension verticale d'une cellule (40 lignes)
      poff  : carré de points -› noeud inactif (0*0 points)
      pon   : carré de points -› noeud actif (3*3 points)
      tcomp : épaisseur des composants (9 points/ligne)
}

procedure INI_CAN;
{     Primitive d'initialisation du canal courant (Sygr) de manière à ce
      que son comportement devienne conforme aux conditions de fonctionne-
      ment de ce module (à utiliser chaque fois que l'on a modifié une
      caractéristique ayant un effet sur le comportement des primitives de
      Sygr pour le canal donnant accès à la représentation graphique).
}

procedure CIRC_VIDE;
{     Primitive initialisant le canal courant pour qu'il contienne un
      circuit vide.
}

{PRIMITIVES D'ACCES A UNE CELLULE}
{Remarque générale : byte de direction(s) construit par Nord, Sud,. }

procedure CRE_BRA(depart : coins; direction : byte; logo : char);
procedure MOD_BRA(depart : coins; direction : byte; logo : char);
{     Primitives de création/modification d'un seul composant.
      Le noeud de référence doit déjà faire partie du circuit.
      La primitive de modification change uniquement le logo.

      depart    : noeud de référence dans la cellule
      direction : localisation à partir du noeud de référence
      logo      : caractère de différenciation des composants
}

procedure CRE_NOD(nref : coins);
procedure SUP_NOD(nref : coins);
{     Primitives permettant de marquer qu'un noeud nref (n')appartient
      (plus) au circuit.
}

procedure MOD_NOD(depart : coins; directions : byte);
{     Primitive permettant d'effacer une ou plusieurs branches partant de
      depart suivant une ou plusieurs directions.
}
IMPLMENTATION {rgra.pas}
```

ANNEXE VI. Accès parallèle

6.1. Présentation générale de PARALAX

La fonction de ce module est d'offrir un certain nombre de primitives réalisées à partir des services offerts par les modules Rext et Rgra.
Les quelques primitives offertes permettent d'avoir une vision homogène de la double représentation du circuit. Il ne s'agit en aucune manière d'offrir un ensemble complet de services mais plutôt d'éviter que chaque module utilisant les deux représentations à la fois dispose d'un certain nombre d'actions prédéfinies ayant un effet logiquement global sur "la" représentation du circuit (vue par l'utilisateur).
Le rôle de ce module est aussi de garantir le fait que le canal d'accès à la représentation graphique du plan soit invariant du point de vue du module Rgra.

Du point de vue de la récupération d'erreurs, une variable *status* indiquera si le service demandé a été totalement exécuté avec succès.
Le service offert par chacune des primitives sera indivisible, autrement dit, il est entièrement réalisé ou il reste sans aucun effet. La seule exception à cette règle concerne les primitives d'aide au stockage qui nécessitent un compactage des données qui, lui, est irréversible.

6.2. Présentation de l'interface

```
unit paralax;
INTERFACE
uses calcul,      {gestion des complexes}
     Sygr,        {gestion envir de Rgra}
     Rext,Rgra,   {les deux représentations accédées en //}
     mix;         {points cardinaux}

var  status : boolean; {vrai = succès}
     canac  : idCan;   {canal d'accès à la représentation graphique}

{
Primitives d'aide à la sauvegarde du contexte invariant -› canal  d'accès à
la représentation graphique.
}

procedure PROCAN;
{     Primitive  délimitant le début d'une période où le canal de la repré-
      sentation graphique peut être modifié.  Enregistrement de l'état cou-
      rant (qui pourra donc être restitué).
}

procedure RSTCAN;
{     Primitive  de  restitution du contexte enregistré  par  procan.  (fin
      d'une  période où le canal de la représentation graphique  peut  être
      modifié).
}
```

```
{Primitives de manipulation globale du plan du circuit}

procedure PLAN_VIDE;
{     A partir des données définissant les caractéristiques de la double
      représentation (cfr Rext, Rgra), cette primitive génère un plan vide
      (pas de noeuds, pas de composants,...).
}

function SAV_PLAN(ref_w : pointer; mxl : longint) : longint;
{     Primitive de sauvegarde de la double représentation du plan selon le
      même mécanisme que SAV_DEF (cfr Sygr).
      ATTENTION : comme elle nécessite un compactage de la représentation
      externe, sa réversibilité n'est pas garantie.
}

procedure RST_PLAN(ref_r : pointer; taille : longint);
{     Primitive de restitution d'un plan sauvé par SAV_PLAN. Il faut
      cependant qu'après avoir généré un plan vide (PLAN_VIDE), ce dernier
      ait des dimensions suffisantes pour recevoir le plan sauvé.
}

{Primitives de modification du plan}

function GOTO_NOEUD(i,j : dim) : coins;
{     Primitive permettant de désigner un noeud. Elle garantit que ce der-
      nier sera visible pour l'utilisateur ainsi que les 8 directions à
      partir desquelles on peut en partir (une cellule entière doit pouvoir
      être visible).
}

procedure ADD_NOEUD(i,j : dim);
{     Primitive d'activation d'un noeud : il devient un élément du plan.
}
procedure SUP_NOEUD(i,j : cim);
{     Primitive de destruction d'un noeud : il ne fait plus partie du plan
      et toutes les branches qui y sont reliées (composants, fils) sont
      également détruites.
}

function ADD_COMPO(i1,j1,i2,j2 : dim; tc : tycmp) : Rcomp;
{     Primitive de création d'un composant reliant deux noeuds.
      Si le composant existait déjà, sa référence est renvoyée et aucune
      modification n'apparaît sauf si tc est différent du type du composant
      préexistant : dans ce cas, il est réinitialisé comme s'il s'agissait
      d'un nouveau composant.
      S'il est impossible de relier les deux noeuds, la référence renvoyée
      est indéfinie et aucune modification n'apparaît.
      S'il s'agit d'un nouveau composant, sa référence est renvoyée et la
      structure de données correspondante n'est initialisée qu'en ce qui
      concerne la référence des deux noeuds (de même en ce qui concerne les
      références des composants des deux noeuds).
      Pour ce dernier cas, une branche au logo (cfr Rgra) correspondant au
      type du composant (tc) apparaît dans la représentation graphique du
      plan.
}
```

```
procedure SUP_COMPO(i1,j1,i2,j2 : dim);
{     Primitive de destruction d'un composant reliant deux noeuds. La
      destruction s'opère au niveau des deux représentations uniquement si
      le composant existe bien dans la représentation externe (cfr Rext).
}

procedure ADD_FIL(i1,j1,i2,j2 : dim);
procedure SUP_FIL(i1,j1,i2,j2 : dim);
{     Primitives de gestion du cablage. (cfr ???_COMPO)}


function MICROSCOPE (xh,yh : corsup; cmp : Pcomp) : idCan;
{     Primitive donnant accès à une fenêtre virtuelle affichée directement
      sur le support en xh,yh. Cette dernière contient un squelette de
      présentation des caractéristiques des composants et si cmp est diffé-
      rent de nil, le squelette sera complété avec les données relatives au
      composant particulier.
}

IMPLEMENTATION {paralax.pas}
```

ANNEXE VII. Manipulation

7.1. Présentation générale de MANIX

Le rôle de ce module est limité à la manipulation de la représentation du plan APRES sa construction. C'est-à-dire après la simulation de son comportement ou au cours de celle-ci pour demander des compléments d'information ou signaler des 'singularités'.

7.2. Présentation de l'interface

```
unit manix;
INTERFACE
uses calcul,mix,Sygr,paralax,Rext,Rgra,Menu;
type cause = (CRT_CIRC,TRP_CMPLX,FOUR,FRIGO);

procedure EXPLORATION;
{     Primitive de gestion de l'exploration du plan par l'utilisateur.
      Celle-ci est limitée aux déplacements et à la consultation des carac-
      téristiques des composants.
}

procedure VINOEUD(i,j : dim; pq : cause; msg : Pstr);
procedure VIBRANC(b : Pcomp; pq : cause; msg : Pstr);
{     Primitives d'aide à la signalisation d'incidents dont la cause est
      pq. En dehors du message mentionnant la cause de l'incident, un
      message msg optionnel peut être ajouté.
}

IMPLEMENTATION {manix.pas}
```

ANNEXE VIII. Construction

8.1. Présentation général de GENIX

Ce module est dédié à la construction de la double représentation du plan du circuit.
Il offre trois fonctions, une primitive d'initialisation d'un plan à laquelle on fournit les dimensions logiques du plan, une primitive de modification de ce plan qui est l'implémentation de l'éditeur de circuit mentionné dans le chapitre trois.
Enfin, il est prévu pour une version ultérieure que l'on puisse greffer des fragments de circuits qui auraient été sauvés par SAV_PLAN (cfr manix).
C'est à ce module qu'incombe le choix de l'objet qui contiendra la représentation graphique du circuit.

8.2. Présentation de l'interface

```
unit genix;
INTERFACE
uses calcul,mix,Sygr,paralax,Rext,Rgra,Menu;

var Rplan   : DREXT;  {descripteur de la représentation externe}
    Caplan  : TypCan; {par défaut VueM d'un écran}
    Maxalim : Rcomp; {0 par défaut -› pas de lim du nbr de sources}

procedure NOUVEX(i,j : dim);
{     Primitive de définition d'un nouveau plan. Si un plan existait encore, il est détruit pour faire place au nouveau plan vide de dimensions i,j (i lignes de j noeuds).
      Le canal d'accès (cfr paralax) est (re)défini ainsi que la représentation externe courante.
}

procedure MODCIR;
{     Primitive "d'édition" du plan du circuit courant. Si ce dernier n'est pas initialisé (NOUVEX), elle est sans effet.
}

function  GREFFE(ref_r : pointer; taille : longint):boolean;
{     Primitive non encore implémentée (tjs FALSE ‹› succès)}

IMPLEMENTATION {genix.pas}
```

ANNEXE IX. Représentation interne

9.1. Présentation générale de RINT

Ce module implémentant la représentation interne du circuit sera basé sur les mêmes principes que ceux qui ont présidé à la définition du module consacré à la représentation externe (cfr Rext : annexe IV).
Néanmoins, dans ce cas, rien n'est prévu pour accéder directement à toute la structure.

9.1.1. Structure de données

La structure de données est le reflet fidèle de la structure décrite dans le chapitre trois. Nous tiendrons compte ici des observations faites au chapitre quatre en ce qui concerne les dimensions des différents éléments qui la composent.

Comme pour le module Rext, les caractéristiques principales de la structure seront rassemblées dans un descripteur.

9.1.2. Primitives de gestion

Les primitives de gestion seront semblables à celles de Rext à la différence près qu'ici on n'insistera que sur le stockage éventuel des résultats de la simulation et non plus sur le stockage de la structure toute entière.
De plus, dans ce cas, il faut offrir toute une série de primitives liées plus étroitement à l'aspect dynamique de la structure.

9.2. Présentation de l'interface

```
unit Rint;
INTERFACE
uses heap,   {allocation de la structure de données}
     calcul; {représentation complexe}


const mxBra = 255; {nombre maximum de branches}
      mxNod = 255; {nombre maximum de noeuds}
      mxStr = 255; {nombre maximum d'étoiles}
      mxTrs = 255; {nombre maximum de branches}
      RintR : byte = 0; {code d'erreur}


type Bref    = 0..MxBra; {type -> référence d'une branche}
     Nref    = 0..MxNod; {type -> référence d'un noeud}
     NuTr    = 0..MxTrs; {type -> identifiant d'une transformation}
     NuSt    = 0..MxStr; {type -> identifiant d'une étoile}

     N2REF   = word; {lo -> N1, hi -> N2}
                     {couple de références et sens du courant 1->2}
     Sref    = word; {type -> référence d'une étoile (offset)}

     tyTrs   = (IDENTITE,SERIE,PARAL,ETRI);
     {type de transformation en fonction de leur complexité 0,1,2,3}
```

```
     Tref    = record {référence d'une transformation}
                      t  : tyTrs; {son type}
                      ad : word;  {localisation dans la pile}
                      end;
     StkTrs  = array[1..MxTrs] of Tref;
     {pile de références des transformations}

     refBra   = string[mxBra];
     {tableau des références des branches actives}
     PrefBra = ^refBra;

     refNod  = string[mxNod];
     {tableau des références des noeuds actifs}
     PrefNod = ^refNod;

     etage   = record ns,nos   : byte
                      {n° de l'étage et n° d'ordre dans l'étage}
                      end;

{branches}
     branch  = record N12    {référence des noeuds et sens du courant}
                      : N2ref;
                      IMPED, {impédance}
                      COUR,  {courant}
                      DDP    {différence de potentiel}
                      : cmplx;
                      end;
     Pbranch = ^branch;

     Bset    = array[1..MxBra] of branch; {ensemble de branches}
     PBset   = ^Bset;

{noeuds}
     node    = record NBra  : Bref;  {nombre de branches connectées}
                      Niv   : etage; {étage du noeud}
                      Etref : Sref;  {référence de l'étoile}
                      end;
     Pnode   = ^node;

     Nset    = array[1..mxNod] of node; {ensemble de noeuds}
     PNset   = ^Nset;

     etoile  = array[1..mxBra] of Bref; {étoile la plus grande}

{Transformation série}
     Tser    = record {avant transformation}
                      AvSt1,AvSt1 : Sref; {ref 2 étoiles modifiées}
                      Nsup        : Nref; {ref noeud supprimé}
                      Bsup1,Bsup2 : Bref; {ref 2 branches sup}
                      {après transformation}
                      ApSt1,ApSt1 : Sref; {ref 2 étoiles mod}
                      NewBr       : Bref; {ref nouvelle branche}
                      end;
```

```
{Transformation parallèle}
     Tpar     = record {avant transformation}
                       AvSt1,AvSt1 : Sref; {ref 2 étoiles modifiées}
                       Bsup1,Bsup2 : Bref; {ref 2 branches sup}
                       {après transformation}
                       ApSt1,ApSt1 : Sref; {ref 2 étoiles mod}
                       NewBr       : Bref; {ref nouvelle branche}
                       end;

{Transformation étoile-triangle}
     Tetr     = record {avant transformation}
                       AvSt1,AvSt2,AvSt3 : Sref; {ref 3 ét. modif.}
                       Nsup              : Nref; {ref noeud sup.}
                       Bsup1,Bsup2,Bsup3 : Bref; {ref 3 br. sup.}
                       {après transformation}
                       ApSt1,ApSt2,ApSt3 : Sref; {ref 3 ét. modif.}
                       NewB1,NewB2,NewB3 : Bref; {ref 3 nvl. br.}
                       end;

     Trs      = record Bigest : Tetr; end;
     {la plus grande transformation -› allocation mémoire}

{état de la structure de données}
     DETAT    = record Nbr_nod : Nref;      {nbr réel de noeuds}
                       Nbr_bra : Bref;      { "    "     branches}
                       Nbr_etl : NuSt;      { "    "     étoiles}
                       SP_etl  : Sref;      {1° place libre -› étoile}
                       Nbr_trs : NuTr;      {nbr réel de transfo.}
                       SP_trs  : Tref;      {1° place libre -› transf}
                       Nactifs : PrefBra;   {@ ensemble des réf -› br}
                       Bractiv : PrefNod;   {@ ensemble des réf -› nd}
                       Limtrs  : ^StkTrs;   {@ pile ref -› transfo.}
                       end;
     REDETAT = ^DETAT;

     DRINT    = record Tot_nod : Nref;      {taille struct -› noeuds}
                       Tot_bra : Bref;      {taille struct -› branch}
                       Tot_str : NuSt;      {taille struct -› étoile}
                       Tot_trs : NuTr;      {taille struct -› transf}
                       Ens_nod : PNset;     {@ ensemble des noeuds}
                       Ens_bra : PBset;     {@ ensemble des branches}
                       Ens_etl : pointer;   {@ ensemble des étoiles}
                       Pil_trs : pointer;   {@ pile des transfo.}
                       ETAT    : REDETAT;   {@ état de la structure}
                       end;

     Pdrnt    = ^DRINT;


var  Simur     :   pointer;
    {@ routine de traitement d'erreur.
     Cette routine n'est pas utilisée par ce module, son rôle est identique
     à  celui de router (cfr Sygr).  Par défaut,  une procédure ne  faisant
     rien  est fournie.  Cette possibilité est offerte pour  permettre  aux
     modules chargés de la simulation (CHACIR, REDUC et PROPAC) de signaler
```

```
     les incidents qui se produisent (erreurs, demandes d'information, fin
     de simulation demandée par l'utilisateur).
    }
     Adcll   : pointer;
    {adresse depuis laquelle une procédure de simulation a été appelée
     (cfr SYGR.adrerr)
    }

{Primitives d'initialisation}

procedure DEF_RIN(var d : DRINT);
{    Primitive d'initialisation. d devient la structure de données courante
     qui est ensuite initialisée.
     En cas d'échec, cette dernière est indéfinie et un code d'erreur est
     généré (rintR).
     Il faut au moins que les tailles des objets soient définies
     (d.tot_????). Si un des pointeurs correspondant (d.ens_???) a la
     valeur nil, la primitive tente de procéder à l'allocation de la mé-
     moire nécessaire. Tout objet dont la référence est différente de nil
     est supposé exister et avoir la taille adéquate (cfr tot_???).
     De plus, si la référence de l'état de la structure est définie, le
     même principe est appliqué concernant chacun de ses éléments (tableaux
     et piles de références). Si par contre, l'état de la structure est
     indéterminé (d.ETAT = nil), il est alors généré conformément aux
     caractéristiques de la structure.
     Enfin, la structure de données est initialisée de manière à repré-
     senter un circuit vide.
}

procedure SUP_RIN;
{    Primitive de destruction de la structure de données courante (resti-
     tution de la mémoire, mise à nil des pointeurs et destruction du
     descripteur de l'état de la structure DETAT).
     Après destruction, la représentation courante est indéfinie.
}

procedure SABRE;
{    Primitive de destruction partielle de la structure de données de
     manière à ce qu'elle ne représente plus que le circuit de départ
     (avant toute transformation).
     L'excédent de mémoire est restitué et dans la mesure du possible, tous
     les éléments de la structure seront consécutifs dans la mémoire (cfr
     code d'erreur rintR si ce n'est pas le cas).
}

procedure GOMERIN;
{    Primitive de réinitialisation de la structure de données. Si la repré-
     sentation courante est définie (cfr DEF_RIN), après GOMRIN, elle
     représentera un circuit vide (ni branches, ni noeuds,...).
}
```

```
{Primitives de gestion (évolution)}
{
Lorsqu'une  erreur survient,  le comportement par défaut des primitives  de
gestion est :
  - le résultat est indéfini
  - la structure de données reste inchangée
  - un code d'erreur est généré
}

procedure CRALIM (IV : boolean; gp : cmplx);
{    Primitive de création de la branche d'alimentation du circuit. Si IV
     a la valeur true,  il s'agit d'une source de courant,  sinon il s'agit
     d'une source de tension.  En fonction de IV,  gp est donc la valeur du
     courant  ou de la tension de cette alimentation.  La référence  de  la
     branche  d'alimentation est toujours 0.  Il faut donc que la structure
     de données représente un circuit vide sinon il y a erreur.
}

function  CREBRAN(z : cmplx) : Bref;
{    Primitive  de création d'une branche d'impédance z.  Le courant et  la
     ddp  sont mis à zero.  En cas de succès,  la référence de  la  branche
     générée  est renvoyée.  Elle ne devient pas une branche active et  les
     noeuds auxquels elle est connectée sont indéfinis.
}

procedure ACTIBRA(b : Bref);
{    Primitive  d'insertion  de la branche b dans l'ensemble  des  branches
     actives.  Si elle en faisait déjà partie,  un code d'erreur est généré
     et aucune modification n'intervient.
}

function  CRETOILE(cnx : refBra) : Nref;
{    Primitive  de  création  d'une étoile et de son centre  (un  noeud)  à
     partir des références des branches qui y sont connectées (cnx). En cas
     de succès, la référence du noeud est renvoyée (sinon résultat indéfini
     et génération d'un code d'erreur).
     L'étage du noeud généré (niv) est indéfini.
     Le noeud généré devient un noeud actif.
     L'ordre des références de cnx est conservé.
}

procedure KILETOILE(Net : Nref);
{    Primitive  de  désactivation d'un noeud Net.  S'il faisait partie  des
     noeuds  actifs,  sa  référence est supprimée de la  liste  des  noeuds
     actifs.  Sinon  aucun changement n'intervient et un code d'erreur  est
     généré.
}
```

```
procedure MODETOILE(Net : Nref; var AvE,ApE : Sref; cut : refBra;
                    nBr : Bref);
{   Primitive de modification d'une étoile de centre Net dont on demande
    de couper les branches cut que l'on désire remplacer par nBr nouvel-
    le(s) branche(s).
    Les variables AvE et ApE contiendront ensuite respectivement la réfé-
    rence de l'étoile avant modification et la référence de l'étoile après
    modification. Les nBr premières références de cette dernière étoile
    seront celles des nouvelles branches générées, les suivantes étant les
    références des branches de l'ancienne étoile que l'on n'a pas coupées.
    En cas de succès, le noeud Net est modifié de telle sorte qu'il
    devienne le centre de l'étoile ApE (Nbra, Etref). Les branches cou-
    pées sont désactivées et les branches générées deviennent des branches
    actives.
    Si l'état de la structure de données ne permet plus de générer une
    nouvelle étoile ou de nouvelles branches, aucun changement n'est
    apporté et un code d'erreur est généré. Dans tous les cas, le contenu
    des variables A?E est toujours défini et est identique si aucune
    modification n'est intervenue (nBr = 0 & cut = '' ou erreur).
}

procedure RETOUR(Net : Nref; AvE : Sref; nBr : Bref);
{   Primitive de restitution d'une ancienne étoile (AvE) comportant nBr
    branches. L'étoile dont le centre est le noeud Net est "oubliée" : les
    branches qui ne font pas partie de l'étoile AvE sont désactivées et
    les branches qui faisaient partie d'AvE sans appartenir à l'étoile
    courante de Net redeviennent actives.
    Le contenu des champs du noeud Net est modifié en conséquence et s'il
    ne faisait plus partie des noeuds actifs, il est réactivé.
}

function CRETRAN(t : Tytrs) : pointer;
{   Primitive de création d'une transformation de type t dont le contenu
    est indéfini et dont l'adresse est renvoyée en cas de succès (si la
    pile des transformations n'est pas remplie).
}

{Primitives d'interrogation}
function LIRTRAN (num : NuTr; var P : pointer) : Tytrs;
{   Primitive d'accès à la transformation num. Si cette transformation
    existe, P contiendra son adresse et son type sera renvoyé. Sinon P est
    indéfini et la valeur IDENTITE est renvoyée. Aucun code d'erreur
    n'est généré.
}

function TELESCOPE(Net : Nref) : refBra;
{   Primitive d'accès au contenu d'une étoile. En cas de succès (net
    défini) elle renvoie la suite des références des branches connectées
    au noeud Net.
}
```

```
function  LIRNOEUD(N : Nref) : Pnode;
function  LIRBRANC(B : Bref) : Pbranch;
{     Primitives  renvoyant  l'adresse des noeuds et des  branches  dont  on
      connait la référence (N,B).  Si l'élément auquel on veut accéder n'est
      pas défini,  la valeur nil est renvoyée mais aucun code d'erreur n'est
      généré.
      Ces  primitives  ne sont à utiliser que lorsque l'on veut  accéder  en
      toute  sécurité  à des branches ou à des noeuds non actifs.  Dans  les
      autres  cas,  un  accès  direct (en utilisant  le  descripteur  de  la
      structure)  est beaucoup plus rapide (calcul d'adresse dans un tableau
      d'éléments de longueur fixe).
}

IMPLEMENTATION {rint.pas}
```

ANNEXE X. Définition d'un circuit

10.1. Présentation générale de CHACIR

Le rôle de CHACIR est d'offrir une série de primitives qui permettent de définir la représentation interne du circuit comme cela a été suggéré dans l'algorithme 3.1 (chapitre trois). A partir des options prises dans le chapitre quatre, ce module a donc une double fonction : créer la représentation interne d'un circuit et le diviser en étages.

10.2. Présentation de l'interface

```
unit chacir;
INTERFACE
uses heap,rint,calcul;
{    Protocole de communication.
     Ce module est basé sur le principe d'un échange partagé d'un certain
     nombre d'informations.
     Pour le rendre indépendant de la représentation externe, il est
     absolument indispensable de pouvoir lui fournir des informations sans
     qu'il sache exactement d'où et comment elles lui parviennent.
     Pour réduire le plus possible le nombre de "choses" que doivent se
     partager ce module et celui qui l'utilise j'ai choisi de définir un
     processus en deux temps :
       - Des signaux de début et de fin de construction (NEWCIR)
       - Des accès indirects et partagés à l'information (les indirections)
}

procedure NEWCIR(nb,nn : byte; var cur,ten : cmplx);
{    Primitive d'initialisation de la représentation interne courante.
     Après avoir réinitialisé la structure de données (en fonction du
     nombre de noeuds (nn) qu'elle doit comporter), newcir génère la bran-
     che d'alimentation sous forme d'une source de courant ou de tension en
     fonction de cur,ten.
     Si cur est nul, alors c'est une source de tension. Si ten est nul
     alors c'est une source de courant. Si les paramètres sont tous les
     deux soit nuls, soit non nuls, un code d'erreur est généré et la
     routine de traitement d'erreur est appelée.
     nb indique le nombre total de branches que le circuit comprend.
     Au retour de cette procédure, la représentation interne sera
     complètement initialisée, représentant ainsi le circuit courant.
}

{
Les variables suivantes sont des pointeurs contenant la référence de pri-
mitives (FAR procedure/function) qui sont susceptibles d'être appelées tant
que NEWCIR n'a pas rendu la main.
Lorsque cette dernière primitive est appelée, cela signifie que toutes les
primitives dont la référence est connue de CHACIR sont prêtes à être
utilisées pour construire la représentation interne d'un nouveau circuit.
Une fois que NEWCIR a rendu la main, elle ne seront plus appelées.
ATTENTION au interférences entre le programme appelant NEWCIR et les
primitives que CHACIR utilise de manière indirecte...
}
```

```
var NewBra : pointer;{:=@procedure prim(var impd : cmplx);}
    {   Tant qu'il existe une branche différente de la source d'alimenta-
        tion et dont on n'a pas encore demandé les caractéristiques, prim
        doit définir l'impédance de celle-ci.
        Il faut également que prim retienne le numéro d'ordre de cette
        branche qui correspond au moment où sa demande de définition a été
        faite considérant que la source est la première branche.
        Cette primitive ne sera plus appelée lorsque les nb branches
        auront été générées.
    }

    GenPos : pointer;{:=@function prim():byte;}
    GenNeg : pointer;{:=@function prim():byte;}
    {   Ces deux primitives doivent permettre de connaître l'identifiant
        des deux bornes (même principe de compostage que pour NewBra mais
        ici on part de 0).
        En plus, elle doivent renvoyer le nombre de branches qui sont
        reliées à la borne dont il est question
    }

    NewNod : pointer;{:=@function prim(num_nod,num_bra : byte):byte;}
    {   Primitive d'identification d'un nouveau noeud, seconde extrémité
        de la branche num_bra (num_nod = 1° extrémité).
        Elle renvoie le nombre de branches qui y sont reliées.
    }

    GenStr : pointer;{:=@function prim(num_nod : byte; var rb : string);}
    {   Cette primitives permet de définir entièrement une étoile (dont le
        centre est le noeud num_nod) en complétant la liste des références
        rd qui peut être vide au départ et qui ne pourra comporter plus
        d'éléments que le nombre annoncé par NewNod.
    }


IMPLEMENTATION {chacir.pas}
```

ANNEXE XI. Réduction d'un circuit

11.1. Présentation générale de REDUC

Le rôle de ce module est de réduire un circuit pour lequel existe une représentation interne.
Il s'agit d'une des deux étapes de simulation d'un circuit. Si on l'a séparé de la seconde (propac), c'est pour pouvoir utiliser cette étape comme un outil autonome.
L'interface de l'unité correspondante se limite à une seule fonction renvoyant la valeur TRUE si la réduction s'est terminée avec succès.
Dans le cas contraire, un code d'erreur aura été généré, mais la routine de traitement d'erreur n'aura pas été appelée.
Comme ce module se base sur l'état courant de la structure de données, en cas de fin prématurée (causée par exemple par un break utilisateur), on peut reprendre la phase de réduction autant de fois que cela est nécessaire à partir de l'endroit où elle s'était arrêtée (Sauf bien-entendu si la réduction s'était terminée avec succès).
En fin de réduction, la structure de données représente une source alimentant un composant unique ainsi que toutes les transformations qui ont permis de trouver ce composant équivalent au circuit de départ.

Les causes de terminaison sont les suivantes :
- Break de l'utilisateur
- Circuit trop complexe (étoiles de plus de 3 branches)
- Erreur de calcul lors d'une transformation (over/underflow)
- Détection d'un court-circuit.
- ....
- réduction réussie

11.2. Présentation de l'interface

```
unit reduc;
INTERFACE
uses calcul,rint,
     Ubreak; {Ubreak.break}

{paramètre -› recupération des erreurs}
var  Tabort : Nutr; {transformation -› échec}

function REDUIRE : boolean;

IMPLEMENTATION {reduc.pas}
```

ANNEXE XII. Propagation des résultats

12.1. Présentation générale de PROPAC

Ce module est l'implémentation de la phase de propagation, complément de la phase de réduction. Il se comporte de la même manière que réduc.
Lorsque la propagation est complètement terminée, l'état courant de la structure de données représente le circuit de départ pour lequel le comportement de tous les éléments constitutifs est connu.

Les causes de terminaison sont les suivantes :
- Break de l'utilisateur
- Erreur de calcul lors d'une transformation (over/underflow)
- Indétermination du sens du courant d'un composant.
- ....
- réduction réussie

Un mécanisme de reprise spécifique permet de lever l'indétermination du sens du courant traversant un composant.

12.2. Présentation de l'interface

```
unit propac;
INTERFACE
uses calcul,rint,
     Ubreak;{Ubreak.break}

{paramètres -› récupération des erreurs}
var  Tcour : NuTr; {transformation courante}
     Cindt : Bref; {branche dont le sens du courant est indéterminé}

function PROPAGER : boolean;

procedure Reprise(N1,N2 : nref);
{    Primitive permettant de lever l'indétermination du sens du courant
     pour un composant (N1--->N2).
}

IMPLEMENTATION {propac.pas}
```

ANNEXE XIII. Gérant d'écran

13.1. Présentation générale de SYGR

SYGR se présente sous la forme d'un gérant d'écran graphique.
Sa particularité réside dans le fait que le texte et les graphiques sont accessibles via les mêmes chemins (une version orientée texte permettrait de recompiler toute application n'utilisant que du texte sans y apporter aucun changement).
En effet, comme c'est le cas avec la version 4 du turbo pascal, le texte et les graphiques sont souvent totalement séparés (CRT,GRAPH) bien que du côté graphique, on retrouve toujours un moyen d'afficher des caractères.

SYGR offre un ensemble de primitives permettant d'accéder à différents types d'objets (Support, fenêtre, écran, fenêtre virtuelle, vue) grâce à un moyen unique : le canal d'accès.

Dans la suite du texte, on considérera comme utilisateur le programme utilisant SYGR.

13.2. Définitions

13.2.1. Objets élémentaires et coordonnées.

Ce module se base sur les services offerts par HAR_CGA, MULTEX et TXDEV. Il permet donc de manipuler du texte (caractères formés par une matrice points) et des points.

Les coordonnées d'un objet situé dans un plan par rapport à un point de référence seront ici exprimées sous la forme d'un couple (coordonnée horizontale, coordonnée verticale).

Un point appartenant à un objet est toujours situé par rapport au coin supérieur gauche de ce dernier. Le coin supérieur gauche (point de référence) aura les coordonnées (0,0).

Un caractère texte appartenant à un objet est toujours situé par rapport au coin supérieur gauche de ce dernier. Le coin supérieur gauche (point de référence) aura les coordonnées (1,1).

Un caractère graphique appartenant à un objet sera toujours situé par les coordonnées du point supérieur gauche de sa matrice de points.

13.2.2. Support
Le support est l'écran physique auquel on accède via hard_cga. Dans le cas qui nous occupe, nous considérerons qu'il s'agit d'un espace dans lequel on peut afficher 25 lignes de 80 caractères, ou 200 lignes de 640 points, les caractères étant formés par une matrice de 8 lignes de 8 points.

13.2.3. Fenêtre
Une fenêtre est définie comme étant une partie du support.
Elle est toujours localisée par les coordonnées dans le support de son coin supérieur gauche.
Sa taille (en nombre de caractères) ne peut dépasser celle du support.

On appelera NIVEAU de fenêtre, un ensemble de fenêtres pouvant être accessibles en parallèle (sans interférence).

Seule une fenêtre peut se voir dotée d'un bord l'entourant complètement. Il existe 8 types de bords prédéfinis ainsi que la possibilité de définir un nouveau type.
Un bord occupe une partie de la fenêtre, restreignant ainsi la zone utile de cette dernière (16 lignes horizontales, 16 colonnes verticales).

13.2.4. Ecran
Un écran est comparable à un support virtuel.
Il est défini par les mêmes éléments que ceux qui caractérisent le support avec comme différences :
- son adresse (pas obligatoirement alignée sur une limite de segment)
- sa taille (en nombre de caractères : max 64k & un seul segment)
- les lignes paires et impaires ne sont plus regroupées
  (représentation logique = représentation physique)

13.2.5. Fenêtre virtuelle
Une fenêtre virtuelle est un écran représentant une fenêtre, elle a donc les caractéristiques de ces deux objets.
Elle permet de conserver en mémoire une 'photo' d'une fenêtre avec la possibilité d'accéder directement à cette 'photo'.

Il y a deux possibilités de rendre visible le contenu d'une fenêtre virtuelle :
- projection de la fenêtre virtuelle dans une fenêtre existante pour autant que cette dernière ait EXACTEMENT la même taille.
- création automatique d'une fenêtre permettant l'affichage de la fenêtre virtuelle.

Une fois qu'une fenêtre virtuelle est rendue visible, l'accès se fait directement sur la projection de son contenu (donc plus directement sur la 'photo' qui devient inaccessible).

Lors de la destruction de la projection de la fenêtre virtuelle sur le support, il y a deux types d'actions possibles :
- mise à jour de la 'photo' en fonction du contenu de la fenêtre.
- aucun des changements survenus n'affectent la 'photo'.

13.2.6. Vue
De manière imagée, une vue est "une fenêtre sur un écran".
Elle permet d'accéder à un écran en faisant apparaître dans une fenêtre les

modifications apportées à l'écran.
Ces 'apparitions' se font selon trois modes différents :
- chaque modification de l'écran est répercutée immédiatement sur la fenêtre (sans intervention du programme utilisateur)
- chaque modification de l'écran détermine une zone dans laquelle celui-ci a été modifié. On mémorise la localisation de la zone et son étendue aussi longtemps que le programme utilisateur n'a pas explicitement demandé une répercussion sur la fenêtre des modifications intervenues depuis la répercussion précédente
- le programme utilisateur demande explicitement l'affichage d'une partie déterminée de l'écran dans la fenêtre

La partie de l'écran affichée a toujours la même taille que la fenêtre dans laquelle elle est affichée (la taille de l'écran >= taille de la fenêtre tant horizontalement que verticalement).

Lorsqu'un canal d'accès permet d'accéder à un écran par une vue, l'accès direct à l'écran sans passer par la vue est impossible.

13.2.7. Canal d'accès

Après avoir défini de manière différente les objets précédents, le seul et unique moyen d'y accéder de manière uniforme est l'ensemble des primitives d'actions sur un canal d'accès (modification, suppression).

On ne peut utiliser un canal d'accès à un objet que si ce dernier est accessible :
- Fenêtre : si elle est "visible", c'est à dire si elle appartient au niveau courant de fenêtres.
- Ecran : s'il n'existe pas de vue projetant une partie de son contenu dans une fenêtre, il est toujours accessible. Si une vue existe, il faut que la fenêtre utilisée soit accessible.
- Fenêtre virtuelle : si elle est projetée sur le support, les conditions sont les mêmes que pour une fenêtre, sinon ce sont celles qui concernent les écrans qui deviennent valables.
- Vue : elle n'est accessible que si la fenêtre utilisée l'est.

Si l'on tente d'accéder à un objet inaccessible, c'est une erreur et aucune modification de l'état courant du gérant d'écran n'a lieu.

Les primitives d'accès se basent sur la notion de canal d'accès COURANT.
Certaines d'entre elles, sont cependant sans effet sur certains objets (cfr [restrictions] dans la description de l'interface).

13.3. Définitions "implicites"

Le support est implicitement défini par des variables initialisées représentant ses caractéristiques. (cfr unit SCRCFG)

Pour chaque objet, on appelera "zone utile", la partie de l'objet qui reste accessible par les primitives d'affichage (cfr DEF_MSQ).

Les primitives d'entrées-sorties standards (read(ln) et write(ln)) sont modifiées selon les principes exposés dans TXDEV.

13.4. Récupération des incidents

13.4.1. User's Break

Pour plus de détails, il est utile de consulter la description de l'unité UBREAK.
Les primitives write(ln) et read(ln) testent l'occurence d'un tel événement en utilisant la primitive BrkChk (cfr Ubreak).
Le traitement de l'événement dépend de la routine dont on a passé l'adresse par l'intermédiaire de On_break. Rappelons que le traitement par défaut est de ne rien faire.

13.4.2. Les erreurs

Lorsqu'une erreur se produit, SYGR place un code d'erreur dans la variable "coder" et l'adresse de l'appel de la primitive qui a entraîné l'erreur dans le pointeur "adrerr". La routine de traitement d'erreur dont l'adresse se trouve dans "router" est ensuite appelée. Un retour éventuel à l'appelant est ensuite généré.
Par défaut la routine de traitement de SYGR ne fait rien.

13.5. Présentation de l'interface

```
unit sygr;
INTERFACE

uses mix,heap,xset,scrcfg,hard_cga,multex,txdev,SygCfg;

const
      {attrib}
      NorV    = $06;   {attribut -› video normale}
      InvV    = $70;   {attribut -› video inverse}

      {atFen : attribut de définition d'une fenêtre (virtuelle)}
      atDef   = $0000; {défintion par défaut}
      Defisl  = $000F; {masque -› définition Fen/Fev                        }
      ClrSup  = $0001; {Effacement du support après définition              }
      VidSat  = $0002; {Attribut video DefVat sinon video normale           }
      FvPhot  = $0004; {Prise de "photo" -› fin de l'affichage d'une Fev    }
      NsvBuf  = $0008; {Pas de sauvegarde du contenu du support (Fen)       }
      Brdisl  = $00F0; {masque -› bords                    }
      BrdSup  = $0010; {si bord ‹› nib : bord supérieur    }
      BrdDrt  = $0020; {                 droit             }
      BrdInf  = $0040; {                 inférieur         }
      BrdGch  = $0080; {                 gauche            }
      QuadBd  = $00F0; {               les quatre bords}
      Ttrisl  = $1F00; {masque -› titre                              }
      TtrInf  = $0100; {Titre sur bord inférieur                     }
      TtrGch  = $0200; {Titre -› gauche                              }
      TtrDrt  = $0400; {Titre -› droite                              }
      TtrCtr  = $0600; {Titre -› centre                              }
      TtrUsr  = $0800; {cfr paramètre x dans def_tit                 }
      RbdBrd  = $1000; {Reconstruction du bord si Nouveau titre}
      Mgrisl  = $E000; {masque -› mode graph}
```

```
      MgrSet  = $2000; {Mode Graphique Set  }
      MgrRst  = $4000; {                reset}
      MgrOr   = $6000; {                or   }
      MgrAnd  = $8000; {                and  }
      MgrXor  = $A000; {                xor  }

      {idCan}
      Badef   = -1; {identifiant de canal pour une définition erronée}
      {idNif}
      Genese  = 0;  {permet de retourner à l'état initial du support}

type  corsup  = word;                 {type -› coordonnées d'un point  }
      pattern = array[0..7] of byte; {matrice de points 8*8           }
      attrib  = byte;                 {attribut video                  }
      idNif   = byte;                 {identifiant -› Niveau de fenêtre}

{bords d'une fenêtre (voir aussi hard_cga.def_brdU}
      Pbord   = ^bord;
      bord    = record csg,           {coin supérieur gauche      }
                       lsh,           {ligne supérieure           }
                       csd,           {coin supérieur droit       }
                       lvd,           {ligne verticale droite     }
                       cid,           {coin inférieur droit       }
                       lih,           {ligne inférieure           }
                       cig,           {coin inférieur gauche      }
                       lvg :          {ligne verticale gauche     }
                             byte     {caractère de construction  }
                       end;

      TypBrd  = (Nib,  {pas de bords}
          {8 types de bords : 4 éléments caractéristiques ci-dessous}

                 Brd0,Brd1,Brd2,Brd3,Brd4,Brd5,Brd6,Brd7,

          {csg : #032 #218 #201 #214 #213 #218 #218 #218  }
          {lsh : #032 #196 #205 #196 #205 #030 #042 #004  }
          {lvd : #032 #179 #186 #186 #179 #179 #179 #179  }
          {lih : #032 #196 #205 #196 #205 #031 #042 #004  }
                 BrdU  {bord défini par l'utilisateur}
                );

{modes graphiques -› primitives graphiques (pas le texte : Gra_set)}
      ModGra  = (Gra_set,  {équivalent mov des,ori       }
                 Gra_reset,{           and des,(not ori) }
                 Gra_or,   {           or  des,ori       }
                 Gra_and,  {           and des,ori       }
                 Gra_xor   {           xor des,ori       }
                );
```

```
{objets manipulables + définitions}
      idCan   = integer; {type de l'identifiant d'un canal      }
                         {objet accédé                          }
      TypCan  = (dirF,   {Fenêtre                               }
                 VueFv,  {Vue -› Fenêtre virtuelle (=Fenêtre)   }
                 dirV,   {Image interne d'une fenêtre virtuelle }
                 dirE,   {Ecran                                 }
                 VueA,   {Vue -› Ecran avec maj automatique     }
                 VueS,   {                      semi-automatique}
                 VueM);  {                      manuelle        }

      atFen   = word;    {attribut de définition d'une fenêtre  }
      DEFN_V  = record   {définitions par défaut -› création    }
                       brd : TypBrd; {le type de bord Fen. (vir)}
                       Tyf : atFen;  {attribut de définition    }
                       end;

      ETAC    = record tyc     : TypCan;  {Type de canal                 }
                       MG      : Modgra;  {Mode Graphique                }
                       atV     : byte;    {Attribut                      }
                       th,tv   : byte;    {épaisseur du trait (hor,ver)  }
                       clic    : boolean; {true si clip -› car entiers   }
                       tcH     : byte;    {taille horiz des car (des_txt)}
                       tcV     : byte;    {       verti                  }
                       xi,yi   : corsup;  {situation initiale de l'objet }
                       xg,yg   : corsup;  {situation actuelle            }
                       xim,yim : corsup;  {   taille initiale            }
                                          {    (0,0) => Etac non valide  }
                       x,y     : byte;    {position courante du curseur  }
                       xm,ym   : byte;    { position extrême             }
                       end;

{variables globales -› gestion des erreurs}
var   coder  : byte;     {code d'erreur}
      router : pointer; {adresse d'une routine de traitement des erreurs}
      adrerr : pointer; {adresse de l'appel à la base de l'erreur}

{variables -› définitions par défaut -› création}
      PredF  : ^DEFN_V; {paramètres par défaut -› définition raccourcie}
      DefatV : attrib;  {attribut à utiliser -› définitions (cfr atFen)}

{}
{primitives de manipulation de matrices de caractères (8x8) [cfr MULTEX]}
{}

procedure DEF_UFO(var carset; tf : taille);
{   Primitive de définition d'un ensemble de caractères affichables unique-
    ment par la primitive DES_TXT.

    carset est la référence de la police de caractères,
    tf définit le type de matrice de points (cfr multex)
}
```

```
procedure DEF_CAR(font : taille; num : byte; var p : pattern);
{   Primitive  permettant de redéfinir la matrice de points d'un  caractère
    d'une des deux polices (cfr hard_cga).

    font  définit la police
    num défini le numéro du caractère (0..255)
    p est la référence d'une matrice de points (8 bytes)
}

{}
{primitives de défintion/manipulation des différents objets}
{}

function  PDF_FEN(xh,yh : corsup; xc,yc : byte) : idCan;
function  DEF_FEN(xh,yh :  corsup;   xc,yc : byte;
                  brd   :  TypBrd; hds,hdi : Pstr;
                  Tyfn  :  atFen):idCan;
{   Primitive de défintion d'une fenêtre renvoyant l'identifiant  d'un
    canal d'accès à la fenêtre en cas de succès (sinon Badef).
    En  cas  de succès,  le canal d'accès courant devient celui de  la
    fenêtre (sinon aucun changement).

    xh,xh : coordonnées du coin supérieur gauche de la fenêtre dans le
            support  (la  fenêtre doit pouvoir s'y  afficher  entière-
            ment).
    xc,yc : taille de la fenêtre en nombre de caractères
                (yc lignes de xc caractères)
    brd   : type de bord (cfr description du type}
            Remarque :  si  brd <> aucun,  la zone utile de la fenêtre
            est reduite à (yc-2) lignes de (xc-2) caractères.
    hds,
     hdi  : string header centré à mettre respectivement dans le  bord
            supérieur et inférieur (uniquement si un bord est défini).
    Tyfn  : masque = (Defisl or Brdisl Mgrisl)

    La  primitive  PDF_FEN  permet de définir une fenêtre  avec  un  nombre
    réduit  de  paramètres.  Dans ce cas,  on utilise pour y  suppléer  les
    données accessibles via PredF et defatV.
}


function  PDF_FEV(xh,yh : corsup; xc,yc : byte) : idCan;
function  DEF_FEV(xh,yh : corsup; xc,yc : byte;
                  brd   : TypBrd; hds,hdi : Pstr;
                  Tyfv  : atFen):idCan;
{   Primitive de définition d'une fenêtre virtuelle renvoyant l'identifiant
    d'un  canal  d'accès  à la fenêtre virtuelle en cas  de  succès  (sinon
    Badef).
    En cas de succès,  le canal d'accès courant devient celui de la fenêtre
    virtuelle (sinon aucun changement).

    xh,xh : coordonnées du coin supérieur gauche de la fenêtre dans le
            support (la fenêtre doit pouvoir s'y afficher entièrement)
            Il  s'agit de l'endroit où la fenêtre virtuelle sera affi-
            chée en cas de visualisation directe sur le support.
```

xc,yc : taille de la fenêtre en nombre de caractères
(yc lignes de xc caractères)
brd : type de bord (cfr description du type)
Remarque : si *brd* <> aucun, la zone utile de la fenêtre est reduite à (yc-2) lignes de (xc-2) caractères
hds,
hdi : string header centré à mettre respectivement dans le bord supérieur et inférieur (uniquement si un bord est défini)
tyfv : masque = (Defisl or Brdisl Mgrisl)
La primitive PDF_FEV permet de définir une fenêtre virtuelle avec un nombre réduit de paramètres. Dans ce cas, on utilise pour y suppléer les données accessibles via PredF et defatV.
}

```
function  DEF_ECR(xc,yc : byte):idCan;
```
{ Primitive de définition d'un écran renvoyant l'identifiant d'un canal d'accès à l'écran en cas de succès (sinon Badef).
En cas de succès, le canal d'accès courant devient celui de l'écran (sinon aucun changement).

xc,yc : taille de l'écran en nombre de caractères
(yc lignes de xc caractères)
}

```
function  DEF_VUE(can_fen,can_ecr : idcan; TyV : TypCan):idCan;
```
{ Primitive de définition d'une vue renvoyant l'identifiant d'un canal d'accès à la vue en cas de succès (sinon Padef).
En cas de succès, le canal d'accès courant devient celui de la vue.
Une vue permet de visualiser le contenu d'un écran ou d'une fenêtre virtuelle.
L'écran doit être visualisé dans une fenêtre déjà définie.
La fenêtre virtuelle peut être visualisée soit dans une fenêtre déjà définie soit directement sur le support (création automatique d'une fenêtre).

can_fen : identifiant d'un canal d'accès à une fenêtre
(0 si visualisation directe d'une fenêtre virtuelle)
can_ecr : identifiant d'un canal d'accès à l'objet visualisé (écran, fenêtre virtuelle)
TyV : type de vue créée
VueFv : vue -> fenêtre virtuelle
VueA : -> écran avec maj automatique
VueS : semi-automatique
VueM : manuelle
}

```
function  NIF_COUR : idNif;
procedure REST_NIF(nif : idNif);
```
{ NIF_COUR renvoie l'identifiant du niveau courant de fenêtres.
REST_NIF restitue l'état d'un niveau antérieur de fenêtres.
(elles sont toutes détruites jusqu'au niveau demandé ainsi que les vues définies à partir des fenêtres détruites).
}

```
procedure ACT_CAN(canal : idCan);
{   Primitive d'activation d'un canal ACCESSIBLE (devient canal courant).}

procedure SUP_CAN(canal : idCan);
{   Primitive de destruction d'un canal ACCESSIBLE.
    S'il s'agit du canal courant ce dernier devient indéfini.
}

function  INF_CAN(canal : idCan; var etat : ETAC):boolean;
{   Renvoie l'état d'un canal (pas nécessairement le canal courant).
            la valeur true si le canal est accessible
}

function  ACC_CAN(canal : idCan):boolean;
{   Renvoie la valeur true si canal est accessible }

procedure DEC_MAJ;
{[uniquement si le canal est du type VueS]
    Primitive  de  déclenchement de la maj semi-automatique de  l'affichage
    d'une vue.
}

procedure MAJ(x1,y1,x2,y2 : byte);
{[uniquement si le canal est du type VueA, VueS, VueM]
    Primitive de maj manuelle de l'affichage d'une vue.
    (x1,y1)  est  le coin supérieur gauche de la  partie  rectangulaire  de
    l'objet à visualiser.
    (x2,y2)  est  le  coin inférieur droit de la  partie  rectangulaire  de
    l'objet à visualiser.
    Il  faut que la zone rectangulaire soit compatible avec les  dimensions
    de la fenêtre et de l'écran.  Elle doit de plus être contenue  entière-
    ment dans ces deux objets (pas de clipping).
}

procedure PHOTOGRAPHIE;
{   Si le canal courant permet d'accéder à une vue,  cette primitive a pour
    effet  de prendre une photo de ce qui est affiché sur le support et  de
    l'envoyer sur l'objet visualisé.
    Pour une fenêtre virtuelle :  effet identique à la maj lors de la  des-
    truction de sa projection.
    Pour  un écran :  projection du contenu de la fenêtre sur la partie  de
    l'écran qui est "visible" à ce moment-là.
}

{}
{primitives d'accès aux différents objets}
{}

procedure DEF_MSQ(xofs,yofs : integer; xc,yc : byte);
{   Permet de définir la zone utile d'un objet (masque d'accès).
    Toutes  les  primitives d'accès fonctionnent dans les limites de  cette
    zone (référence, clipping).
}
```

```
function  MOD_CAN(xh,yh : corsup; Tyc : TypCan) : boolean;
{[ssi le canal courant permet d'accéder à une fenêtre (virtuelle)]
    Modification de ses coordonnées dans le support.
    Attention : ce type d'action est impossible pour une fenêtre créée avec
    l'attribut NsvBuf ou une fenêtre utilisée dans des vues.
 [ssi le canal courant permet d'accéder à une vue sur écran]
    Modification du mode de maj d'une vue sur écran.

    xh,yh sont nuls pour exprimer la valeur courante.
}

procedure DEF_BRD(brd : TypBrd; hds,hdi : Pstr);
{   Permet  de  définir un bord à une fenêtre  (virtuelle)
    [cfr def_fen pour les paramètres]
    Remarque :  il y a effacement ou non de la zone utile en fonction de la
    définition de l'objet (attribut, ClrSup).
}

procedure DEF_TIT(x : word; md : atFen; ps : Pstr);
{   Permet de définir un titre dans un bord.
    Le bord ne pourra être reconstruit que s'il a été défini  par
    def_brd ou lors de la définition de l'objet.

    x :  emplacement  du  titre  par  rapport  à la  limite  gauche  de  la
         zone utile
    md : masque (Ttrisl)
    ps : cfr hds,hdi (def_fen)
}

procedure GOTOXY(x,y : byte);
{   Modifie la position du curseur texte.
    (Sans effet si les coordonnées sortent des limites de la zone utile)
}

function  whereX : byte;
function  whereY : byte;
{   Renvoie la position courante du curseur texte (x,y)}

function  XtxMax : byte;
function  YtxMax : byte;
{    Renvoie la position extrême que peut atteindre le curseur texte}

function  XgrMax : corsup;
function  YgrMax : corsup;
{   Renvoie la position extrême d'un point (x) sur une ligne
                              ou d'une ligne (y) graphique
    dans l'objet courant
}

procedure DEF_MG(m : ModGra);
function  GET_MG : ModGra;
{   Mode graphique selon lequel l'objet sera modifié
    DEF -› définition
    GET -› renvoie le mode courant.
}
```

```
procedure DEF_AT(at : attrib);
function  GET_AT : attrib;
{    Attribut des caractères de texte (video normale ou inverse).
     DEF -› définition
     GET -› renvoie le mode courant.
}

procedure XOR_AT(x,y,nx : byte);
{   Inverse l'attribut des nx caractères de la ligne y à partir de  la
    colonne  x  et  ce quel que soit le mode graphique  ou  l'attribut
    courant.
}

procedure CLRSCR;
procedure CLREOL;
procedure CLRBOL;
procedure CLRLINE;
{   Primitives d'effacement (attention à l'attribut courant).
    ClrScr  :
       - efface toute la surface de l'objet
       - remet le curseur texte en (1,1)
    ClrEol  :
       - efface  depuis la position courante du curseur texte  jusqu'à
         la fin de la ligne courante
       - sans effet sur la position du curseur
    ClrBol  :
       - efface depuis la position courante du curseur texte  jusqu'au
         début  de la ligne courante
       - sans effet sur la position du curseur
    ClrLine :
       - efface la ligne courante
       - remet le curseur texte en (1,ligne courante)
}

procedure INSLINE(en_dessous : boolean);
{   Insère une ligne vide au dessus ou en dessous de la ligne courante.
    (Attention au scrolling  de  la zone utile)
    Place le curseur texte au début de la ligne insérée.
}

procedure DELLINE(par_dessous : boolean);
{   Supprime  la ligne courante en la recouvrant avec la ligne du dessus ou
    la ligne du dessous.
    (Attention au scrolling  de  la zone utile)
    Place le curseur texte en (1,ligne couvrant la ligne supprimée)
}

procedure TXCLIP(on : boolean);
function  CLIP : boolean;
{   Gestion du clipping à la limite des caractères.
    TXCLIP -› Définition du mode
    CLIP   -› Vrai si affichage de caractères graphiques entiers uniquement
}
```

```
procedure AFCAR(c : char);
{   Primitive affichant un seul caractère 'c' à la position courante du
    curseur texte, sans modifier celle-ci (cfr write(c) sans aucun autre
    effet que l'affichage du caractère).
}

procedure FILLCAR(c : char; x,y,nc,nl : byte);
{   Permet de remplir une zone rectangulaire de la zone utile à l'aide
    d'un caractère.
    c      : numéro du caractère
    x,y    : caractère supérieur gauche du rectangle
    nc,nl  : taille du rectangle en nombre de caractères (horiz,vertic)
    Cette primitive est affectée par l'attribut et le fait que l'on permet-
    te ou non l'affichage partiel de caractères.
}

procedure DEF_FNT(mtx_H,mtx_V,lvs,lvi : byte);
{   Primitive de définition du type de caractère à utiliser -› des_txt.
    mtx_H,mtx_V : taille de la matrice de points
    lvs,lvi     : nombre de lignes vides à ajouter au dessus et en dessous.
}

procedure DES_TXT(x,y : word;s : string);
{   Dessine le contenu de s en x,y
    Cette primitive est affectée par :
         - le type de caractère (def_fnt)
         - l'attribut video et le mode graphique
         - possibilité ou non de l'affichage partiel de caractères
}

procedure FILLPATTERN(var p : pattern;xd,yd,xpix,ylin : corsup);
{   Permet de remplir une zone rectangulaire de la zone utile à l'aide d'un
    pattern de points.
    p          : référence de la matrice de points
    xd,yd      : point supérieur gauche du rectangle
    xpix,ylin  : taille du rectangle en nombre de points (horiz,vertic)
    Cette primitive est affectée par le mode graphique.
}

function  TAILLE_PIC(x1,y1,x2,y2 : corsup):word;
procedure GET_PIC(x1,y1,x2,y2 : corsup; var buffer);
procedure PUT_PIC(x,y : corsup; var buffer);
{   Primitives de manipulation de fragments rectangulaires d'images.
    Taille_Pic
    renvoie la taille (en bytes) que doit avoir un buffer pour
    pouvoir contenir un rectangle dont le coin supérieur gauche se situe en
    (x1,y1) et le coin inférieur droit en (x2,y2).
    Get_Pic
    place le contenu de l'image rectangulaire (cfr taille_pic) dans un
    buffer.
    Put_Pic
    restitue le contenu d'un buffer dans une zone rectangulaire de l'image
    située aux coordonnées (x,y).
    REMARQUE : en plus de l'image proprement dite, le buffer contiendra la
    description de la taille de celle-ci.
```

```
}

function  GET_PIX(x,y : corsup) : boolean;
{   Vrai si le point est allumé (! épaisseur du trait : toujours 1,1) }

procedure DEF_ET(ep_h,ep_v : byte);
{   Définit l'épaisseur du trait (horiz,vertic) -› draw et plot. }

function ETH : byte;
function ETV : byte;
{   Renvoie l'épaisseur courante du trait (Horiz, Vertic). }

procedure PLOT(x,y : corsup);
{   Affiche un point.
    Cette  primitive  est affectée par le mode graphique et l'épaisseur  du
    trait limitée à 8 lignes de 8 points.
}

procedure DRAW(x1,y1,x2,y2 : corsup);
{   Trace une droite.
    Cette  primitive est affectée par le mode graphique et  l'épaisseur  du
    trait.
}

{}
{Gestion de la sauvegarde -› fichiers}
{}

function SAV_DEF(ref_w : pointer; mxl : longint) : longint;
{   Fonction  permettant  de  sauver la définition du canal  courant  comme
    objet  d'une collection (cfr COLLECT) qui renvoie la taille de  l'objet
    sauvé.
    Les vues ne peuvent être sauvées.
    L'image liée à la définition est sauvée séparément.
    Si une erreur survient, la fonction renvoie une taille d'objet nulle.

    ref_w : référence  de la fonction  permettant d'écrire dans la  collec-
            tion dans laquelle on veut sauver l'objet.
            [far F_W(adr mem, # bytes) : boolean]
    mxl   : taille max que peut avoir la définition (-1 si aucune limite)
}

function RST_DEF(ref_r : popinter; taille : longint): idCan;
{   Fonction  permettant  de charger la définition d'un  canal  sauvée  par
    SAV_DEF.
    En  cas  de succès,  elle renvoie l'identifiant d'un canal d'accès  qui
    devient  le  canal courant.  (erreur :  cfr erreur de  définition  d'un
    objet).

    ref_r  : référence de la fonction permettant de lire dans la collection
             dans laquelle se trouve l'objet.
             [far F_R(adr_mem, lot : word) : boolean]
    taille : taille de la définition (en bytes)
}
```

```
function SAV_IMG(ref_w : pointer; mxl : longint) : longint;
function RST_IMG(ref_r : pointer; taille : longint) : boolean;
{   Fonctions (cfr ???_DEF) permettant de gérer le stockage des images
    liées au canal courant.
    RST_IMG ne renvoie pas l'identifiant d'un canal d'accès, seulement la
    valeur true en cas de succès.

    Remarque : pour une fenêtre virtuelle, la 'photo' fait partie de
    sa définition. L'image n'existe que lorsqu'il y a projection directe de
    cette 'photo' sur le support
}

{Remarque générale.
Les fonctions SAV_??? et RST_??? font explicitement référence à une collec-
tion d'objets (cfr gestion de fichier). Rien n'empêche cependant d'utiliser
des fichiers classiques en implémentant deux fonctions F_R/W à l'aide des
procédures prédéfinies BlockRead et BlockWrite.
}

{}
IMPLEMENTATION {sygr.pas}
{}
```

ANNEXE XIV. Gestion de Menus

14.1. Présentation générale de MENU

Ce module offre une gestion automatique des menus utilisant comme entrée un clavier virtuel et en sortie une gestion de fenêtre fournie par SYGR. Le traitement des erreurs utilise également les variables qui lui sont allouées dans SYGR (coder, router, adrerr : seul coder est utilisé directement par ce module).

Les caractères issus du clavier virtuel sont définis à partir des codes fournis par l'interruption 16h.
Par défaut, ce clavier a la fonctionnalité de BDOS.kbd.

14.2. Défintions

14.2.1. Les champs d'un menu

Un menu est défini comme un ensemble de champs répartis dans une image. On considérera ici que ces champs sont rangés en lignes horizontales comportant toutes le même nombre de champs (sauf la dernière si c'est nécessaire). La répartition sur une ligne est aussi régulière, en ce sens que tous les champs comportent le même nombre de colonnes et que l'espace entre eux est toujours le même (tant horizontalement que verticalement).
Si *Gch* est le nombre de colonnes entre les champs d'une même ligne et que *GchL* est le nombre de lignes vides séparant deux lignes de champs, alors la référence (curseur texte de Sygr) du premier caractère du premier champ par rapport au coin supérieur gauche de l'image sera (*Gch*+1,*Gchl*+1).
De plus, le nombre de champs devra être compris entre 1 et 255.

14.2.2. L'image d'un menu

La représentation des champs d'un menu sera une image contenue dans une fenêtre (virtuelle) dont les caractéristiques feront partie de la définition du menu (cfr paramètres de définition d'une fenêtre (virtuelle) dans Sygr).
Le contenu de cette fenêtre sera défini de deux manières, d'une part l'image que l'utilisateur voit et d'autre part, la répartition des champs dans cette image.
Pour ce qui est de l'image, la référence d'une procédure sans argument sera passée comme paramètre de la définition d'un menu. Cette procédure sera chargée de dessiner le menu grâce aux primitives d'accès au canal courant (cfr Sygr). Ce dernier sera défini, avant l'appel de cette procédure, comme la fenêtre (virtuelle) devant contenir l'image.
En règle générale, le rôle de cette procédure sera de définir les bords éventuels, la zone utile (cfr Sygr) et le contenu des différents champs.

14.2.3. Les différents types de menus

En plus des modes d'interprétation des commandes (voir ci-dessous), il y a trois caractéristiques qui peuvent affecter le comportement d'un menu.
Premièrement, il faut savoir si le menu sera affiché par l'intermédiaire d'une fenêtre virtuelle ou non. Si l'on choisit de l'afficher par une fenêtre virtuelle, la procédure de construction de l'image ne sera appelée qu'une seule fois et le menu pourra être sauvé dans un fichier. Par contre, si l'on choisit une fenêtre, à chaque affichage du menu, ce dernier sera entièrement reconstruit : il ne pourra donc être sauvé dans un fichier (il

est impossible de sauver la procédure de définition).
Enfin, faut-il ou non faire disparaître automatiquement l'image du menu lorsqu'on en sort et si non, faut-il dans ce cas laisser visible la marque du dernier champ sélectionné ?
Ces trois caractéristiques se retrouvent traduites par les constantes *Photo*, *ReMnu* et *ReChp* (cfr interface).

14.2.4. Interprétation des commandes

Les commandes venant du clavier sont de deux types, d'une part les déplacements du curseur et d'autre part les lettres et les fonctions.
En ce qui concerne les déplacements du curseur, il est important de savoir si les menus sont circulaires et si oui, dans quelle direction.
En effet, un déplacement horizontal du curseur qui demande le dépassement des limites des choix représentés sera interprété en fonction de cette caractéristique (si le menu est circulaire, on revient à l'autre bout de la ligne, sinon on l'interprète comme une commande de sortie de menu). Il en va de même en ce qui concerne les déplacements verticaux.

L'utilisateur aura différentes manières de se faire comprendre :
- déplacement du curseur et touche Return pour marquer son choix (flèches uniquement)
- entrer une lettre représentant directement un choix
- entrer une fonction représentant directement un choix
- entrer une lettre ou une fonction ayant une signification connue du programme appelant la gestion du menu.

Les commandes de l'utilisateur seront répercutées de la manière suivante : code de la lettre ou code de la fonction plus un identifiant. Les deux codes sont des bytes (respectivement le byte inférieur et le byte supérieur du mot renvoyé par l'interruption 16h).
On prendra donc la convention suivante :
- $00##h avec ## = numéro de champs
- $01##h avec ## = code de lettre
- $02##h avec ## = code de fonction
- $03##h avec ## = numéro de commande externe

En ce qui concerne les lettres ou fonctions représentant directement ou non un choix, on fournira trois tables de conversions sous la forme de trois strings. Celles-ci seront optionnelles.
Ces trois tables auront les caractéristiques suivantes :
- La première sera une suite de codes de lettres correspondant aux différents numéros de champs. Cette suite sera continue (pas de trou) mais ne devra pas couvrir obligatoirement l'ensemble des champs.
  Ces codes de lettres auront la forme suivante :
  $01## = caractère ## sans aucun traitement
  et pour les caractères de 0 à 128
  $05## = caractère ## majuscule ou minuscule
  $09## = caractère ## majuscule/minuscule ou caractère de contrôle
  (un striw)
- La seconde représentera une suite continue de codes de fonctions.
  (un string)
- La troisième sera une suite de codes n'ayant aucun rapport direct avec les champs de l'image.
  (un striw : $01##, $02##, $05##, $09##).

Une commande correspondant aux deux premières tables sera traduite en un numéro de champs correspondant. Les ordres reconnus à partir de la troisième table seront quant à eux restitués sous la forme suivante : (300h or *numero de commande externe*) avec le numéro de commande externe égal au numéro d'ordre du couple (1..255).
Le code 0 signale une sortie anormale dont la cause peut être déterminée par coder.
En cas de menus non circulaires, le code de la flèche dans la direction du débordement est renvoyé ($02##, avec ## le code correspondant à une touche curseur).

14.3. Présentation de l'interface

```
unit menu;
INTERFACE
uses bdos,heap,SygCfg,SyGr,mix,strw;

const Mdef    =  0;  {par défaut : aucun des 5 suivants}
      ReMnu   =  1;  {reste affiché => SSP_MNU manuellement}
      ReChp   =  2;  {en sortie le champ reste marqué à l'image}
      Photo   =  4;  {image construite dans une fenêtre virtuelle}
      EscH    =  8;  {Menu non circulaire horizontalement}
      EscV    = 16;  {Menu non circulaire verticalement}

      BdMnu   = -1;  {idMnu en cas d'échec}

type  TypMnu = 0..31;
      {combinaison des valeurs Mdef,ReMnu,ReChp,Photo,EscH,EscV}
      idMnu  = integer;
      Rmnu   = word; {choix de l'utilisateur (cfr convention)}


var  clavier : pointer;
     {référence de la fonction d'entrée (far) simulant le fonctionnement de
      la fonction de BDOS.kbd
     }


procedure SUP_MNU(id : idMnu);
{     Primitive de destruction du menu id, si ce dernier est affiché, et
      que d'autres menus le sont aussi mais pas au même niveau de fenêtre,
      cela va entraîner leur suspension (ainsi que toute fenêtre superposée
      et ayant des interférences avec celle du memu).
}


procedure SSP_MNU(id : idMnu);
{     Lorsque l'on décide qu'un menu reste affiché après que l'on en soit
      sorti, cette primitive permet de l'effacer du support. Comme il
      s'agit d'une destruction de fenêtre, si d'autres fenêtres ont été
      définies à un niveau supérieur, elles seront également détruites.
}
```

```
function  DEF_MNU(xf,yf : corsup; xc,yl : byte; att : TypMnu;
                  car : Pstriw; fct : Pstr; kdir : Pstriw; build : pointer;
                  atf : atfen; MXC,NL,NchL,Lch,Gch,GchL:byte):idMnu;
{     Fonction de définition d'un menu qui renvoie l'identifiant de ce
      menu en cas de succès (sinon BdMnu).
      Cette fonction définit également une fenêtre (virtuelle) qui en cas
      de succès devient le canal courant (sinon ce dernier est indéfini).

      xf,yf : position de la fenêtre du menu (coordonnées).
      xc,yl : taille de la fenêtre du menu (colonnes,lignes)
      att   : type de menu (cfr description du type)
      car   : striw de correspondance lettres-champs
      fct   : string de correspondance fonctions-champs
      kdir  : striw de lettres/caractères reconnus
      build : référence d'une procédure (far) de construction de
              l'image du menu dans la fenêtre du menu.
      atf   : mode d'initialisation de la fenêtre (cfr def_fen)
      MXC   : Nombre total de champs
      NL    : Nombre de lignes de champs
      NchL  : Nombre de champs par ligne de champs
      Lch   : Nombre de colonnes d'un champ
      Gch   : Nombre de colonnes entre champs
      GchL  : Nombre de lignes entre lignes de champs
}


function  REV_MNU(id : idMnu; xh,yh : corsup; df : byte) : Rmnu;
{     Primitive d'activation du menu id renvoyant l'ordre de l'utilisateur
      en cas de succès (sinon 0).
      Si le menu réactivé était déjà affiché, il doit être accessible
      (sinon échec). De plus, dans ce cas les paramètres ci-dessous sont
      ignorés.

      xh,yh : localisation du menu dans l'espace du support.
              ($FFFF -> coordonnées par défaut)
      df    : numéro du champ marqué par défaut.
              (0 -> dernier champ de la fois précédente)
}

function SAV_MNU(ref_w : pointer; mxl : longint; id : idMnu):longint;
function RST_MNU(ref_r : pointer; taille : longint):idMnu;
{     Primitives de gestion de sauvegarde de menus dans un fichier (pour
      les paramètres cfr Sygr).
      Seuls les menus dont l'image apparaît dans une fenêtre virtuelle
      peuvent être sauvés.
      La sauvegarde d'un menu comprend l'enregistrement des éléments
      suivants :
           - Sa définition (y compris les trois tables de conversion si
             elles existent)
           - La définition de la fenêtre virtuelle associée (et donc la
             'photo' associée)
      SAV_MNU ne détruit ni ne suspend le menu sauvé.
}

IMPLEMENTATION {menu.pas}
```

ANNEXE XV. Gestion de "Collections"

15.1. Présentation générale de COLLECT

Ce module permet de gérer des collections (cfr définition chap. 3,4).
Les identifiants des collections, des types d'objets et des noms d'objets comporteront maximum huits caractères qui doivent figurer parmi les caractères permis pour les noms des fichiers MS-DOS.
La gestion se base essentiellement sur la notion de collection courante et d'objet courant dans une collection. En plus des primitives classiques de gestion de fichiers, COLLECT offre des primitives permettant un partage des opérations de lecture et d'écriture dans une collection afin de permettre qu'un programme demande à un module de sauver un objet dont la structure est inconnue du programme.

15.2. Présentation de l'interface

```
unit collect;
INTERFACE

uses bdos,heap,bfil,xfil,hfil,fcbs;
const MxCol  = 16;        {nbr max de collections ouvertes simultanement}
      MxObj  = 255;       {nbr max d'objets/collection}
      Undef  = $FF;
      ColEr  : byte = 0; {code d'erreur}
      atcre  : byte = 0; {code de création de fichier de collection}

type  idCol   = 0..UNDEF;     {identifiant d'une collection ouverte}
      ident   = string[8];    {nom -> objets (type,nom) / collection}
      codesc  = record typ,nom  : ident;  taille  : longint;  end;
                 {descripteur d'objet : type, nom, taille en bytes}

var   Radr    : pointer; {adresse de l'appel -> erreur}
      RHndl   : pointer;
      {référence de la routine de récupération d'erreur (par défaut ne
       fait rien)
      }
      pcur    : path; {par défaut = ''}
                      {sinon toujours terminé par '\'}
                {DOIT être le même pour TOUTES les collections ouvertes}

{
Références des procédures d'écriture et de lecture partagées. Ces références pointent vers une procédure booléeenne qui renvoie la valeur FALSE tant que l'opération partagée n'a pas été initialisée.
}
      RdiW  : pointer; {ref F_W(var admem; siz : word) : boolean}
     {La fonction F_W permet l'écriture d'un morceau d'objet de taille
      siz se trouvant à l'adresse admem à la position courante de la
      collection courante.
      Elle renvoie la valeur TRUE en cas de succès, sinon FALSE.
     }
      RdiR  : pointer; {ref F_R(var admem; siz : word) : boolean}
     {Même fonction que F_W mais en lecture cette fois}
```

```
procedure RstCol;
{     Ferme toutes les collections ouvertes.
}

function  CreC(nom : string) : idCol;
{     Crée une collection nom (max 8 caractères) et renvoie son identi-
      fiant. En cas de succès, cette collection devient la collection
      courante. Si celle-ci existait déjà auparavant, cette dernière est
      détruite (sauf s'il s'agit d'un collection déjà ouverte : dans ce
      cas, elle devient la collection courante)
}

function  OpnC(nom : string) : idCol;
{     Ouvre la collection nom (max 8 caractères) et renvoie son iden-
      tifiant. En cas de succès, cette collection devient la collection
      courante. Si la collection n'existe pas, elle est créée. Si elle est
      déjà ouverte, l'effet est identique à celui d'une sélection.
}

procedure ClsC(col : idCol);
{     Ferme une collection. Si celle-ci était la collection courante, cette
      dernière devient indéfinie.
}

procedure Select(id : idCol);
{     Sélection de la collection courante : si id n'est pas un identifiant
      valide, cette procédure est sans effet.
}

function  DiWr(nobj,tobj : string) : longint;
function  DiRd(nobj,tobj : string) : longint;
{     Primitives d'initialisation d'une opération d'écriture ou de lecture
      partagée relative à l'objet de type tobj et de nom nobj dans la
      collection courante. Si ce dernier n'existe pas, en écriture, il est
      créé (en lecture c'est une erreur). En écriture, la primitive renvoie
      la taille maximale de l'objet, en lecture, la taille réelle de
      l'objet.
      En cas de succès TOTAL de l'opération, l'objet accédé devient l'objet
      courant de la collection (sinon ce dernier est indéfini).
}

procedure WriC(nobj,tobj : string; var admem; taille : longint);
function  ReaC(nobj,tobj : string; var admem; tamax : longint):longint;
{     Primitives d'accès à un objet en écriture ou en lecture. L'objet
      identifié par son type tobj et son nom nobj doit pouvoir se trouver
      en un seul bloc en mémoire à l'adresse admem. En lecture, tamax
      exprime la taille maximale de l'objet tandis que la taille réelle de
      celui-ci est renvoyée.
      En cas de succès, l'objet accédé devient l'objet courant de la
      collection (sinon ce dernier est indéfini).
}
```

```
procedure FiDW(taille : longint);
{     Procédure de confirmation d'écriture partagée. Si la taille de l'ob-
      jet est nulle, l'opération d'écriture est annulée et l'objet est
      détruit.
}


function  Exist(nobj,tobj : string): boolean;
function  Yatil(tobj : string):word;
{     Primitives de localisation d'un objet déterminé (EXIST) ou d'un
      groupe d'objets de même type (YATIL).
      Pour la première, en cas de succès, la valeur TRUE est renvoyée et
      l'objet trouvé devient l'objet courant de la collection courante.
      Pour la seconde, en cas de succès, le nombre d'objets est renvoyé
      (sinon 0) et le dernier devient l'objet courant de la collection
      courante.
      En cas d'échec, l'objet courant est indéfini.
}

function Suiv : boolean;
function Prec : boolean;
{     Primitives permettant de sélectionner comme objet courant un objet de
      même type que celui de l'objet courant mais qui dans le répertoire
      des objets lui succède (suiv) ou le précède (prec).
      En cas de succès la valeur TRUE est renvoyée, en cas d'échec, c'est
      la valeur FALSE et l'objet courant est indéfini.
}

procedure infoc(var dc : codesc);
{     Primitive permettant d'acquérir des informations concernant l'objet
      courant de la collection courante.
}

procedure modif(nobj,tobj : string; var admem; taille : longint);
{     Primitive de modification d'un object existant (cfr écriture).
      Si nobj,tobj ne permet d'identifier aucun objet de la collection, il
      y a remplacement de l'objet courant par le nouvel objet. En cas de
      succès de l'opération (modification/remplacement), il devient l'objet
      courant de la collection courante (sinon ce dernier est indéfini).
}

procedure supri;
{     Primitive de destruction de l'objet courant de la collection
      courante.
}

IMPLEMENTATION {collect.pas}
```

ANNEXE XVI. Accès au hardware vidéo

16.1. Présentation générale

Cette unité permet d'accéder à une carte vidéo dont les caractéristiques doivent être voisines des particularités d'un carte CGA : la mémoire vidéo doit être alignée sur le début d'un segment et ne peut dépasser 64 Kbytes avec des lignes d'au plus 256 bytes. Les lignes paires et impaires doivent être séparées en deux blocs et à chaque bit correspond un point (=› monochrome).

Ce module permet d'accéder directement au hardware compatible CGA
Aucun contrôle n'est effectué concernant la validité des arguments
On considère deux modes d'accès : texte & graphique.

Les objets manipulés sont fonction du mode utilisé
TEXTE : des caractères (+ éventuellement leur attribut)
GRAPH : des points
rangés en lignes à partir du coin supérieur gauche de l'écran
On situe ces ELEMENTS par rapport au coin supérieur gauche de l'écran
Les coordonnées s'expriment sous la forme d'un couple (x,y : word)
avec x : la position horizontale de l'élément
(extrémité gauche = 0)
y : la position verticale de l'élément
(extrémité supérieure = 0)
En mode graphique, on situe la position d'un caractère par les coordonnées du point supérieur gauche de la matrice de points qui le constitue

Ce module d'accès a deux particularités :
- Il permet d'utiliser un jeu de primitives identiques pour les deux modes graphiques.
- Il permet d'accéder à la mémoire vidéo mais aussi à une zone mémoire (buffer) se comportant comme un écran virtuel. Ce dernier a la structure LOGIQUE de la mémoire vidéo (+ fonctions de tranfert entre les deux).

16.2. Interface de HARD CGA

```
unit hard_cga;
INTERFACE
uses  bdos,mix,
      ScrCfg;   {ce  mini  module  permet d'adapter celui-ci  à  différents
                 types de cartes
                }
type  Tcursor   = record xc,yl : byte end;  {curseur texte}

      w8        = array[0..7] of Tcursor;   {8 curseurs texte}

      unidata   = record larB : byte;       {largeur                    }
                         homB : pointer;    {position du buffer courant}
                         end;
```

```
{DEFINITIONS relatives au hardware}
const {control ports references}
      modSel_Rg = $04; {DISP_CPRT + modSel_Rg => mode select register port}
      clrSel_Rg = $05; {DISP_CPRT + clrSel_Rg
                                  => color select register port}
      Status_Rg = $06; {DISP_CPRT + Status_Rg => Status register port}
      LP_latch  = $07; {DISP_CPRT + LP_latch => Light Pen latch port}
      {mode select bits values : CTRL_CODE}
      MTX80_25  = $01; {40*25 => 0}
      MGRAPH    = $02; {320/640 * 200 graphic mode}
      Vid_BW    = $04; {B/W mode}
      Scr_ON    = $08; {off => 0}
      MG_HiRes  = $10; {if MGRAPH => 640}
      Blnk_Vid  = $20; {blink mode}
      MVunused  = $C0; {unused bits}
      {color select bits values : DISP_PALL}
      CS_blue   = $01; {}
      CS_green  = $02; {colors -> foreground, background, border}
      CS_red    = $04; {}
      Cintense  = $08; {intensity bit}
      CP_altIn  = $10; {alternate intensified palette}
      CP_nbr1   = $20; {palette n° 1, 0 => 0}
      {unused bits : cfr MVunused}
      {status bits}
      NoSnow    = $01; {memory acces doesn't generate snow}
      LPtrigOn  = $02; {Light pen triggrer set}
      LP_OFF    = $04; {Light pen off, 0 => On}
      Vretrace  = $08; {vertical retrace}
      Sunused   = $F0; {unused bits}

var   IO_CONFIG : word absolute $0000:$0410; {cfr int 11h}
      DISP_MODE : byte absolute $0000:$0449; {current video mode}
      DISP_COLS : word absolute $0000:$044A; {display width in columns}
      DISP_LENG : word absolute $0000:$044C; {video memory length}
      DISP_STAR : word absolute $0000:$044E; {offset in video memory seg}
      PAGE_CURS : w8   absolute $0000:$0450; {cursor pos / page}
      CURS_MODE : word absolute $0000:$0460; {cursor aspect}
      ACTV_PAGE : byte absolute $0000:$0462; {current video page (0..?)}
      DISP_CPRT : word absolute $0000:$0463; {display ctrl port address}
      CTRL_CODE : byte absolute $0000:$0465; {hardw code -> DISP_MODE}
      DISP_PALL : byte absolute $0000:$0466; {current palette}

{DEFINITIONS relatives aux primitives d'accès}
const {élément d'effacement de l'écran}
      clear     : lh = (l:$20;  {caractère}
                        h:$06); {attribut}
                  {en mode graphique, il n'est tenu compte que du fait
                   que ce soit en mode vidéo directe ($07) par un and ($07)
                                                 inverse ($70)
                  }

      {mode graphique -> (??G?? primitives}
      Gset      = $00;
      Grst      = $01;
      Gor       = $02;
```

```
      Gand      = $03;
      Gxor      = $04;
var   VRP       : minpack;  {-> bdos.screen}
      Hardesc   : ^unidata; {-> descripteur INTERNE (ne peut être modifié)}
      bufInCode : pointer;  {-> buffer de 1024 bytes}

      {Les  2 pointeurs ci-dessous sont définis uniquement si USES_GRA (cfr
       plus loin)}
      normcar   : pointer;  {-> 256 matrices 8*8}
      minicar   : pointer;  {-> 256 matrices 6*4
                               (partie utile dans coin sup gauche 8*8)}
      Present   :  byte;    {1=>code de tx présent,
                             2=>code de gr présent,
                             0=>aucun code présent
                            }

procedure USES_TXT;
procedure USES_GRA;
{     Ces primitives permettent de passer du mode texte au mode graphique.
      Avant  d'utiliser une des primitives ci-dessous,  il faut définir  le
      mode utilisé AU MOINS une fois au début du programme.
}


{}
{primitives communes aux deux modes -> texte/graphique}
{}


function Pic_siz(nx,ny : word) : word;
{     Renvoie la taille d'une image de ny lignes constituées de nx éléments
      En  mode  graphique  chaque ligne est constituée de (nx  /  8)  bytes
      arrondis à l'unité supérieure.
}

function ptr_pic(x,y : word) : pointer;
function ptr_buf(x,y : word) : pointer;
{     Renvoie  la  position du byte contenant l'élément ayant  (x,y)  comme
      coordonnées soit dans l'écran (pic) soit dans le buffer courant (buf)
}

procedure pic_1U(var piof,linum : word);
procedure pic_1D(var piof,linum : word);
{     Primitives permettant d'accéder à la structure logique de l'écran
         piof  constitue l'offset d'un élément dans l'écran
         linum est la ligne dans laquelle se trouve cet élément

      pic_1U  a  pour effet de modifier piof et linum de façon à  permettre
      l'accès  à l'élément de même coordonnée horizontale mais à  la  ligne
      (linum - 1)

      pic_1D est semblable à pic_1U mais pour la ligne (linum + 1)
}
```

```
procedure picTObuf(xo,yo,xd,yd,nx,ny : word);
procedure bufTOpic(xo,yo,xd,yd,nx,ny : word);
procedure bufTObuf(b1,b2 : pointer; lb1,lb2,xo,yo,xd,yd,nx,ny : word);
procedure picTOpic(xo,yo,xd,yd,nx,ny : word);
{     Primitives  de transfert entre écran (pic) et buffer (buf) ainsi  que
      de buffer à buffer.
         (xo,yo) : coordonnées  origine  du  coin  supérieur  gauche  du
                   rectangle à transférer
         (xd,yd) : coordonnées  du  coin  supérieur  gauche  du  rectangle
                   destination
         (nx,ny) : taille du rectangle à transférer
         lb?,b?  : définition des buffers
      Lorsque le buffer n'est pas explicitement défini (lb?,b?)
      hardesc^ est utilisé.
}

procedure picSWbuf(xo,yo,xd,yd,nx,ny : word);
procedure bufSWbuf(b1,b2 : pointer; lb1,lb2,xo,yo,xd,yd,nx,ny : word);
{     Primitives  d'échange entre écran (pic) et buffer (buf) ainsi que  de
      buffer à buffer.
      Voir ???TO??? pour la description des paramètres.

      REMARQUE : Aucun test n'est effectué sur la situation des rectangles
      Le  transfert se fait donc toujours de gauche à droite et de haut  en
      bas
}

procedure fill_pic(car,x,y,nx,ny : word);
procedure fill_buf(car,x,y,nx,ny : word);
{     Primitives de remplissage d'écran ou de buffer
         x,y   : coin supérieur gauche du rectangle à remplir
         nx,ny : taille du rectangle à remplir
         car   : un caractère & son attribut (cfr clear)
}

procedure SCRLuP(nls,x,y,nx,ny : word);
procedure SCRLdP(nls,x,y,nx,ny : word);
procedure SCRLuB(nls,x,y,nx,ny : word);
procedure SCRLdB(nls,x,y,nx,ny : word);
{     Primitives de scrolling d'écran (????xP) ou de buffer (????xB)
         nls   : nombre de lignes d'éléments -› scrolling
         x,y   : coordonnées  du coin supérieur gauche de la zone  affectée
                 par le scrolling.
         nx,ny : taille de la zone affectée par le scrolling
      Le caractère CLEAR est utilisé pour l'effacement
}

procedure UNIC_PIC(car,x,y : word);
procedure UNIC_BUF(car,x,y : word);
{     Primitives d'affichage d'un caractère unique -› (écran/buffer)
         car : caractère & attribut (cfr clear)
         x,y : coordonnées -› affichage
}
```

```
procedure HORC_PIC(car,noc,x,y : word);
procedure HORC_BUF(car,noc,x,y : word);
procedure VERC_PIC(car,noc,x,y : word);
procedure VERC_BUF(car,noc,x,y : word);
{     Primitives d'affichage de noc caractères identiques -› (écran/buffer)
      en ligne horizontale (HORC_???) ou vericale (VERC_???)
         car : caractère & attribut (cfr clear)
         noc : nombre de caractères
         x,y : coordonnées du premier caractère
                   les suivants sont soit à droite, soit en dessous
}

procedure STR_PIC(s : string; x,y : word);
procedure STR_BUF(s : string; x,y : word);
{     Primitives d'affichage d'un string de caractères -› (écran/buffer)
      L'attribut utilisé est celui du caractère CLEAR.
         x,y : coordonnées du premier caractère
}

procedure TXT_PIC(lt : byte; t : pointer; x,y : word);
procedure TXT_BUF(lt : byte; t : pointer; x,y : word);
{     Primitives d'affichage d'une chaîne de caractères -› (écran/buffer)
      L'attribut utilisé est celui du caractère CLEAR.
         lt  : taille de la chaîne de caractères
         t   : référence du premier caractère de la chaîne
         x,y : coordonnées du premier caractère
}

procedure def_brdU(var bc);
function  get_brd(b : byte):pointer;
{      Primitives de manipulation des définitions de bords de rectangles.
       La définition d'un bord est constituée d'une suite de caractères
              coin supérieur gauche
              ligne supérieure horizontale
              coin supérieur droit
              ligne verticale droite
              coin inférieur droit
              ligne inférieure horizontale
              coin inférieur gauche
              ligne verticale gauche
      def_brdU permet de définir un bord particulier
          (bc = @ d'une définition)
      get_brd permet d'obtenir la référence d'un bord (pré)défini
          0    : bord vide (blancs)
          1..7 : 7 bord prédéfinis (cfr get_brd.asm)
          8    : bord défini par def_brdU (0 si non défini)
}

{}
{Primitives particulières au mode texte (80*25)}
{}

type patr = pointer; {masque de modification d'attribut}
```

```
function fpatr(pa,px,po : byte):patr;
{     Fonction  de conversion de trois attributs -› masque de  modification
      d'attribut
         pa : attribut -› and (attribut courant)
         px : attribut -› xor (attribut courant)
         po : attribut -›  or (attribut courant)
}

procedure SnowChk(On : boolean);
{     Primitive de définition de la manière d'accéder à la mémoire vidéo
      Si (On = true) alors pendant le "Vertical retrace time"
         sinon n'importe quand
}

procedure MOD_AT_P(p : patr; nx,x,y : word);
procedure MOD_AT_B(p : patr; nx,x,y : word);
{     Primitives de modification d'attribut sur une ligne d'écran (P) ou de
      buffer (B)
         p   : masque de modification
         x,y : coordonnées du 1° caractère dont il faut modifier l'attribut
         nx  : nombre de caractères dont il faut modifier l'attribut sur la
                  ligne
      Les trois opérations se font dans l'ordre suivant : AND XOR OR
}

procedure CUR_TOG(On : boolean);
{     Primitive permettant de supprimer/restituer le curseur -› écran
}

procedure CUR_ABS(x,y : word);
{     Primitive permettant de définir la position du curseur -› écran
}

{}
{primitives particulières au mode graphique (640*200)}
{}

{
LES PRIMITIVES ????G???? sont affectées par le mode graphique
}
procedure ref_mode(m : byte);
{     Primitive de définition du mode d'affichage graphique affectant  uni-
      quement les primitives dont le nom comprend un G (???G???)
      Pour les autres primitives le mode est toujours Gset
}

procedure movsl(ori,des : pointer; xo,xd,nx : word);
procedure movslG(ori,des : pointer; xo,xd,nx : word);
{     Primitive de transfert d'une ligne de points contigus
         ori : référence du byte contenant le premier point origine
         des : référence du byte contenant le premier point destination
         xo  : position du premier point dans le byte ori (0..7)
         xd  : position du premier point dans le byte des (0..7)
         nx  : nombre de points à transférer
}
```

```
procedure swapL(ori,des : pointer; xo,xd,nx : word);
{     Primitive d'échange d'une ligne de points contigus
         ori : référence du byte contenant le premier point origine
         des : référence du byte contenant le premier point destination
         xo  : position du premier point dans le byte ori (0..7)
         xd  : position du premier point dans le byte des (0..7)
         nx  : nombre de points à échanger
      REMARQUE : le transfert se fait toujours de gauche à droite
}

procedure stosl(des : pointer; pat,xd,nx : word);
procedure stoslG(des : pointer; pat,xd,nx : word);
{     Primitives de remplissage d'une ligne de points contigus par un
      pattern de 8 points (G = affectée par le mode graphique)
         des : référence du byte contenant le premier point à "remplir"
         xd  : position du premier point dans le byte des (0..7)
         nx  : nombre de points à "remplir"
         pat : pattern de 8 points (le 1° -> des(xd))
}

procedure fillGpic(car,x,y,nx,ny : word);
procedure fillGbuf(car,x,y,nx,ny : word);
{     Primitives de remplissage d'écran ou de buffer affectées par le mode
      graphique sinon identiques à fill_???.
}

procedure picTGbuf(xo,yo,xd,yd,nx,ny : word);
procedure bufTGpic(xo,yo,xd,yd,nx,ny : word);
procedure bufTGbuf(b1,b2 : pointer; lb1,lb2,xo,yo,xd,yd,nx,ny : word);
procedure picTGpic(xo,yo,xd,yd,nx,ny : word);
{     Primitives de transfert affectées par le mode d'accès, sinon identi-
      ques à ???TO???
}

procedure xor_at_P(nlx,nx,x,y : word);
procedure xor_at_B(nlx,nx,x,y : word);
{     Primitives de modification (XOR) de l'attribut d'une ligne de points
         nlx : nombre de lignes de points à modifier
         nx  : nombre de points
         x,y : coordonnées du premier point
}

procedure def_tck(H,V : byte);
{     Primitive de définition de l'épaisseur du trait pour le dessin
      (cfr les 4 primitives ci-dessous AFFECTEES par le mode d'accès)
         H : épaisseur horizontale du trait (1 par défaut)
         V : épaisseur verticale du trait   (1 par défaut)
}

procedure pnt_pic(xp,yp : word);
procedure pnt_buf(xp,yp : word);
{     Primitive de dessin d'un "point" débutant aux coordonnées (xp,yp) et
      s'étendant horizontalement sur H points (def_tck)
                 verticalement sur V points (def_tck)
}
```

```
procedure lin_pic(xo,yo,xd,yd : word);
procedure lin_buf(xo,yo,xd,yd : word);
{     Primitive de dessin d'une droite reliant les points de coordonnées
      (xo,yo) et (xd,yd)
      L'épaisseur de la droite est fonction des arguments passés grâce à
      def_tck
}

procedure LI8x8(carset : pointer; ls : word; t : pointer);
procedure LI6x6(carset : pointer; ls : word; t : pointer);
procedure LI6x4(carset : pointer; ls : word; t : pointer);
{     Primitives de génération d'une ligne GRAPHIQUE correspondant au
      texte t de longueur ls (le résultat -> bufincode^ dont le 1° byte
      indique le nombre de bytes que comprend la ligne générée).

      Pour générer toute la matrice de points, il faut utiliser ces primi-
      tives autant de fois que la matrice comporte de lignes.
      A chaque fois, il suffit d'incrémenter l'offset de la matrice de
      point correspondant au premier caractère (carset).
}


IMPLEMENTATION {hard_cga.pas}
{txtHcga.asm, graHcga.asm, comHcga.asm, li8x8.asm, li6x6.asm, li6x4.asm,
 car8x8.asm, car6x4.asm, get_brd.asm}
```

ANNEXE XVII. Extension des primitives texte

17.1. Présentation générale

Ce petit module permet de générer du texte formé de caractères de taille différentes à partir de deux ensembles de caractères (hard_cga.normcar & hard_cga.minicar).
Les caractères générés sont envoyés à l'écran en mode graphique.
Leur attribut est celui du caractère hard_cga.clear.

17.2. Interface de MULTEX

```
unit multex;
INTERFACE
uses mix,hard_cga,scrcfg;

const ls     : byte = 0; {nombre de lignes vides au-dessus  du caractère}
      li     : byte = 0; {                        en-dessous             }

var   clip   : record nx,ny : word; end;
               {
               taille max d'une ligne de caractères (en pixels)
               (0,0) => aucun contrôle de clipping
               }

type  taille = (t6x4,    {caractères formés de 6 lignes de 4 points}
                t6x6,    {                     6          6       }
                t8x8,    {                     8          8       }
                tLxP);   {                     L          P       }

procedure SET_FONT(f : taille; mH,mV : byte);
{     Définition du type de caractère formé (par défaut t8x8).
      mH et mV sont ignorés dans tous les cas sauf pour f=tLxP. Dans ce
      cas, ils donnent respectivement le multiplicateur horizontal et ver-
      tical de la taille des caractères.
      Cette dernière s'obtient comme suit :
         horizontalement => 8 * mH
         verticalement   => (8+ls+li) * mV
}

procedure grP_TXT(lt : byte; t : pointer; x,y : word);
procedure grP_STR(s : string; x,y : word);
procedure grB_TXT(lt : byte; t : pointer; x,y : word);
procedure grB_STR(s : string; x,y : word);
{     Primitives d'affichage de caractères à l'écran (grP_???) ou dans un
      buffer (grB_???). Les caractères se trouvent soit sous la forme d'un
      string (s de gr?_STR) soit sous la forme d'un morceau de texte de lt
      caractères dont le premier est situé à l'adresse t.
      Le premier point du premier caractère est affiché en x,y.
}

IMPLEMENTATION {multex.pas}
{to_CDB.asm, grP_TXT.asm, grB_TXT.asm, grP_STR.asm, grB_STR.asm}
```

ANNEXE XVIII. Input-Output dans un contexte général

18.1. Présentation générale

Ce module permet d'utiliser les primitives prédéfinies du pascal READ(LN), WRITE(LN) dans le cadre d'un programme gérant lui_même les accès à l'écran. Il est entièrement compatible avec le module hard_cga.
Ce dernier est d'ailleurs pris comme référence dans certaines définitions ci-dessous mais rien n'empêche d'utiliser d'autres primitives que les siennes.

18.2. Interface de TXDEV

```
unit txdev;
INTERFACE
uses bdos,mix,Ubreak;

const CR     = -32768;{position extrême gauche du curseur -› MOVCUR}
      NO_MAX = $FFFF; {pas de contrôle de read(ln) par MAX_IN}
      MAX_IN : word = NO_MAX; {nombre max de caractères -› read(ln)}

var   UNICAR : pointer; {@ far hard_cga.unic_pic/buf}
      MULCAR : pointer; {@ far hard_cga.txt_pic/buf}
      MOVCUR : pointer; {@ far procedure movexy(x,y : integer);}
      FIT_IN : pointer; {@ far function fit(ltx : word) : word;}
      PAR_XY : pointer; {^ record x,y : word end;}
      PCLR   : pointer; {^ hard_cga.clear}


IMPLEMENTATION {txdev.pas}
```

18.3. Protocole de communication

Le protocole de communication entre TXDEV et son environnement est le suivant (y compris les spécifications des primitives) :

- Le contrôle du user_break se fera par l'intermédiaire des primitives offertes par UBREAK. Lors de toute entrée-sortie, un 'BrkChk' sera effectué.
- Les entrées de texte se font via bdos.kbd
  Un user break signalé par bdos.kbd sera interprété comme un caractère $0D
- L'affichage de caractère se fera selon la procedure suivante :
  (1) appel à fit_in pour connaître le nombre de caractères contigus affichables sur la ligne courante à partir des coordonnées PAR_XY
  (2) affichage des ce(s) caractère(s) par uni/mulcar
  (3) SI TOUS les caractères sont affichés, mise à jour de la position du curseur par movcur(+ltx,0), SINON passage à la ligne par movcur(CR,+1) et retour au point (1)
- MOVCUR
  Les paramètres passés à cette procédure sont des offsets par rapport à la position courante du curseur (-› integer).
  La valeur CR permet de remettre le curseur à sa position extrême gauche.
  Cette procédure est donc aussi chargée de la gestion du dépassement des limites de la "fenêtre courante" (scrolling, passage à la ligne automatique,...)
  Après l'appel de cette procédure, les valeurs des paramètres PAR_XY reflètent la nouvelle situation du curseur.
- FIT_IN
  Cette fonction renvoie le nombre de caractères (‹= ltx) que l'on peut encore afficher à partir de la position courante du curseur (PAR_XY). Elle permet donc de gérer le clipping.
- PAR_XY
  Paramètres reflètant la position courante du curseur directement exploitable -› paramètres de uni/mulcar

ANNEXE XIX. Gestion des calculs

19.1. Présentation générale de CALCUL

Comme je l'ai mentionné dans le chapitre quatre, la version 4 du turbo pascal n'est pas vraiment une réussite du point de vue des possibilités de calcul sur des nombres réels. C'est pourquoi, il est absolument indispensable d'enfermer dans un module toutes les primitives utilisant directement des opérations sur des réels. En effet, cela facilitera par exemple l'introduction d'améliorations dans ce domaine (changement de librairie de fonctions mathématiques par exemple).
Il existe cependant une seconde raison pour déléguer à un module particulier la responsabilité des calculs : nous allons utiliser des nombres complexes qui n'existent pas directement comme type de base. Il est donc utile de créer de nouvelles fonctions permettant de manipuler globalement de tels nombres.
Enfin, le turbo pascal n'offre aucun moyen de contrôle des erreurs qui peuvent se produire (division par 0, overflow, underflow,...) et qui, à chaque fois, provoquent la terminaison brutale du programme. Il faut donc également offrir un moyen de contrôler ces erreurs.

Le module calcul offrira donc trois types de primitives :
- des opérations sur des nombres complexes
- la conversion complexe-texte, et inversement
- un mécanisme de récupération des erreurs

19.2. Zone de calcul

Le mécanisme de récupération des erreurs de calcul sera basé sur la protection d'une 'zone' de calculs.
Un 'zone' de calculs est une partie d'un programme dans laquelle l'occurence d'une erreur de calcul (d'ordinaire fatale) pourra être récupérée.
Pour délimiter une zone de calculs, on utilisera deux primitives (protect et fihot) décrites ci-dessous.
La structure d'une 'zone' de calculs aura l'aspect suivant :

```
{déclarer le label sortie au début du bloc contenant la 'zone'}
          .
          .
          .
          PROTECT;
          if (MATER <> 0) then goto sortie;
          .
          .
          {'zone' de calculs}
          .
          .
sortie : FIHOT;
          .
          .
          .
```

19.3. Présentation de l'interface

```
unit calcul;
INTERFACE

type   reel  = real; {permet de changer le type de base}
       cmplx = record r,i : reel; end; {représentation rectangulaire}
       rcisA = record m,a : reel; end; {représentation polaire}

 const mater : byte = 0; {code d'erreur -> opération math}

{}
{quatre opérations de base}
{}

procedure ADD(var res,c1,c2 : cmplx);   {res:=c1+c2}
procedure SUB(var res,c1,c2 : cmplx);   {res:=c1-c2}
procedure PROD(var res,c1,c2 : cmplx);  {res:=c1*c2}
procedure QUOT(var res,c1,c2 : cmplx);  {res:=c1/c2}

{}
{opérations de 'conversion'}
{}

function  MODULE(var c : cmplx) : reel;

procedure RP(var cr : cmplx; var cp : rcisA);
{   rectangulaire -> polaire}
procedure PR(var cp : rcisA; var cr : cmplx);
{   polaire -> rectangulaire}

{conversions texte-réel}
function  txSymR(s : string) : réel;
{   primitive de conversion texte -> reel (valeur de composants)
         le texte peut avoir la forme suivante :
              [nombre réel]([blanc(s)][symbole][blanc(s)])[texte]
         avec [nombre réel] : cfr nombre réel valide -> readln
              [blanc(s)]    : au moins un blanc
              [symbole]     : préfixe multiplicatif
              [texte]       : suite (vide) de caractères
              (.....)       : '.....' optionnel
    les préfixes multiplicatifs reconnus sont les suivants :
         p : pico
         n : nano
           : micro
         m : milli
         k : kilo
         M : mega
         G : giga
    Attention,  les  majuscules  et  minuscules  sont  considérées  comme
    différentes
}
```

```
function  RtxSym (var r : reel) : string;
{   Primitive  inverse de txSymR mettant les nombres sous la forme  suivan-
    te : (-)[nnnnnn].[ddddddddddd][blanc][préfixe multiplicatif]
    Le  nombre de chiffres n sera toujours le plus petit possible en  fonc-
    tion du préfixe multiplicatif utilisable.
    Le  nombre de décimales d sera toujours le plus grand possible en fonc-
    tion de la précision de la représentation des réels.
}

function RtxE (var r : reel) : string;
{   Primitive  de  conversion d'un réel en sa  représentation  en  notation
    'exponentielle'
}

{}
{primitives de protection d'une zone de calculs 'à risques'}
{}

procedure protect;
{   Primitive délimitant le début d'une zone de calculs.
    Elle  utilise  un  certain  nombre  de  variables  système  (ExitProc,
    ExitCode, ErrorAddr).
    Après  avoir  appelé cette procédure,  CHAQUE  occurence  d'une  erreur
    (Runtime error) aura l'effet suivant :
       - les  registres SS,SP et BP reprennent la valeur qu'ils avaient  au
         moment de l'appel de cette procédure.
       - MATER prend une valeur différente de 0.
       - le  programme reprend comme s'il revenait de l'appel de la  procé-
         dure  protect (nécessité d'un test pour  différencier l'occurrence
         d'une erreur du retour de la procédure protect).
    Pour éviter toute catastrophe (crash système,...), il est impératif que
    FIHOT  soit immédiatement appelée juste après la détection d'une erreur
    MAIS AUSSI dès que la zone de calculs a été exécutée.
    Remarque :  en cas de Stack overflow, si l'appel à fihot génère la même
    erreur,  le  programme  prendra fin immédiatement.  Si le stack  a  été
    détruit  (au-dessus  de SS:SP mémorisé),  c'est la  catastrophe  (crash
    système) !
}

procedure fihot;
{   Primitive délimitant la fin d'une zone de calculs.
    En  cas  d'erreur,  son rôle est également de récupérer les erreurs  de
    calculs (runtime errors 106,  200,  205,  206 et 207).  Dans les autres
    cas, le programme se terminera.
    Après fihot,  mater une des valeurs suivantes :  0 (pas d'erreur), 106,
    200, 205, 206 ou 207 (cfr runtime errors).
    Remarque :  il  n'y a pas d'interférence avec le contrôle  des  erreurs
    d'entrée-sortie (options $i+/-).
}

IMPLEMENTATION {calcul.pas}
```

AVERTISSEMENT

Pour ceux qui voudraient en savoir plus en ce qui concerne les services que proposent les quatre modules ci-dessous, il leur suffit de consulter le DOS TECHNICAL REFERENCE MANUAL et si une information partielle est suffisante, le NORTON GUIDE (MASM) permet de disposer de cette information "on-line".
En consultant les sources, ils trouveront les paramètres exacts qui sont utilisés pour générer les différents appels système sous-jacents aux primitives présentées ci-dessous.
La gestion des erreurs se fait en accédant à bdos.xcod

ANNEXE XX. Fonctions de base

20.1. Présentation générale

Ce module permet d'accéder aux fonctions de base de gestion de fichier offertes par le système.
Il n'y a absolument aucun contrôle d'erreur prévu si ce n'est le garnissage d'une variable indiquant la manière dont s'est déroulée la dernière opération.
Pour éviter les nombreux passages de paramètres toujours identiques, toutes les primitives se basent sur la notion de fichier courant.

20.2. Interface de BFIL

```
unit bfil;
INTERFACE
uses bdos,mix;


type  path = string[64]; {string terminé par le caractère null }
      ppat = ^path;      {utiliser mix.Astr pour les constantes}


const fcat   : word = 0; {attribut de création de fichier}

      curdrv = '@';      {caratère identifiant le drive courant}

     {attributs -> fichiers             }
      atRO    = $01;      {read-only     }
      atHd    = $02;      {hidden        }
      atSy    = $04;      {system        }
      atVl    = $08;      {volume label  }
      atDr    = $10;      {sub-directory }
      atAr    = $20;      {archive-bit   }
      AllF    = $37;
```

```
     {modes d'ouverture de fichier }
      Mrd    = $00;      {read-only }
      Mwr    = $01;      {write-only}
      Mrw    = $02;      {read/write}

{variables globales -> objet courant          }
var  curDta : pointer;   {dta courante        }
     curFH  : word;      {file handle courant}

procedure select(d : char);
function  cur_dsk : char;
{    Primitives de gestion du drive courant.
     select  -> selection du drive courant (d = A..?)
     cur_dsk -> renvoie l'identifiant du drive courant
}

function  loc_driv : byte;
function  byte_clus(d : char):word;
function  clus_free(d : char):word;
{    Primitives de gestion des "aspects quantitatifs" des unités d'io
     loc_driv  -> renvoie le nombre de drives locaux
     byte_clus -> renvoie le nombre de bytes par clusters pour le drive d
     clus_free -> renvoie le nombre de clusters libres pour le drive d
}

procedure mkdir(p : path);
procedure rmdir(p : path);
procedure chdir(p : path);
procedure get_dir(var p : path; d : char);
{    Primitives de gestion des directories
     mkdir -> création d'une directory, p en est le path complet
     rmdir -> destruction d'une directory dont p est le path
     chdir -> sélection  de  la directory courante du drive dont  l'identi-
              fiant figure en tête du path p (sinon du drive courant)
}

procedure fcreate(p : path);
procedure fopen(p : path; mode : byte);
procedure fclose;
{    Primitives de gestion du fichier courant (CurFH).
     fcreate -> création d'un nouveau fichier ayant pour attribut atCre.
     fopen   -> ouverture d'un fichier selon un certain mode
     Ces deux primitives définissent en cas de succès le fichier courant.
     fclose  -> fermeture du fichier courant (curFH devient indéfini)
}

function  fread(buf : pointer; nb : word):word;
function  fwrite(buf : pointer; nb : word):word;
{    Primitives de lecture/écriture (fread/fwrite) dans le fichier courant
     buf : emplacement mémoire à lire/écrire
     nb  : nombre de bytes à transférer
     Ces primitives renvoie le nombre de bytes effectivement transférés  (0
     si l'opération a échoué)
}
```

```
procedure fseek(w : longint);
function  fpos:longint;
function  fosize:longint;
{    Primitives de gestion de la position courante dans le fichier courant.
     fseek  -› définition de la position courante
               depuis le début du fichier
     fpos   -› renvoie la position courante
     fosize -› renvoie la taille du fichier courant
}

procedure fdelete(p : path);
{    Primitive  de  destruction du fichier p (ce ne peut  être  le  fichier
     courant qui ne subit aucun changement).
}

procedure frename(old,new : path);
{    Primitive  permettant  de  rebaptiser ou de changer  de  directory  UN
     fichier  (ce  ne  peut  être le fichier courant  qui  ne  subit  aucun
     changement).  Les  deux path doivent être complet (pas de ?/*) et doi-
     vent se situer sur le même drive.
}

function  fsize(pp : ppat):longint;
{    Cette  primitive  renvoie la taille du fichier pp (ce ne peut être  le
     fichier courant)
}

function  get_att(p : path):word;
procedure set_att(p : path; attrib : word);
{    Primitive  de gestion de l'attribut du fichier p différent du  fichier
     courant.
     get_att -› renvoie l'attribut du fichier
     set_att -› définit l'attribut du fichier (sauf atDr/atVl)
}

function  dupfh:word;
procedure fdup(master,slave : word);
{    Primitives de duplication de "file-handle"
     dupfh -› crée un frère jumeau à curFH
              -un close du frère jumeaux =› écriture physique
              -permet de sauver un état avant un shell
     fup   -› force le file-handle slave à pointer vers le même fichier que
              master (permet d'effectuer des redirections).
}

function  sfirst(fn : path; at : word):boolean;
function  sNext:boolean;
{    Primitives de recherche de fichier (utilisent curDTA)
     sfirst -› initialisation de la recherche
               fn : masque de recherche (?/* uniquement dans le nom)
               at : attribut de recherche
     sNext  -› APRES sfirst,  cherche le fichier suivant qui correspond aux
               critères définis par sfirst.
}
IMPLEMENTATION  {bfil.pas, bfil.asm}
```

ANNEXE XXI. Accès à la structure physique

21.1. Présentation générale

Ce module offre un certain nombre de définitions qui permettent d'accéder à la structure logique des disques sous MS-DOS.
Dans la présentation de l'interface, j'ai gardé la terminologie technique se référant aux éléments décrits.
Les définitions accompagnées d'une étoile ne sont pas celle du MSDOS et permettent de définir des découpes physiques du disque dur.

21.2. Interface de XFIL

```
unit xfil;
INTERFACE
uses bdos,mix;

{Informations -› disque dur définies seulement si nb_dr ‹› 0}
const MAX_CY : word = 615;   {max number of cylinders       }
      MAX_H  : byte = 4;     {max number of heads           }
      MAX_S  : word = 25;    {max number of sectors/track   }
      SCYL   : word = 100;   {max number of sectors/cyl     }
      HD_ID  : char = 'C';   {first hd drive identifier     }
      NB_DR  : byte = 0;     {number of hd drives           }

type  decomp  = (L,H,S,CY);              {low,hight/head,sector,cylinder}
      dsk_pos = array[H..CY] of byte;    {master boot record format}
      sector = array [0..511] of byte;
{Hard disk Partition record definition}
      partt  = record boot    : byte;    {boot partition signal : $80}
                      pbegin  : dsk_pos;{@ first partition sector}
                      system  : byte;    {system indicator : dos = $01}
                                         {simulated partition number (*)}
                      pend    : dsk_pos;{@ last partition sector}
                      rel_sec,           {logical number of first sector}
                      tot_sec : longint;{partition size in # sectors}
                                 {8 char hidden partition name (*)}
                      end;
{Hard disk Master Boot Record}
      mastr  = record c       : array[0..$0FF] of byte;
                      {code : -› boot}
                      mother  : byte;
                      {(*)mother partition number pp[1..4]}
                      e       : array[$101..$13D] of byte; {fill bytes}
                      p       : array[1..8] of partt;
                      {(*)hidden partitions}
                      pp      : array[1..4] of partt;
                      {system partitions}
                      signature : word;                    {$55AA}
                      end;
```

```
{Bios Parameter Block}
     bpb    = record bps : word;  {bytes/sector}
                     spc : byte;  {sectors/cluster}
                     res : word;  {# reserved sectors}
                     nf  : byte;  {# fats}
                     dir : word;  {# root dir entries}
                     tot : word;  {# sectors (all !)}
                     med : byte;  {media descriptor byte}
                     spf : word;  {sectors/fat}
                     spt : word;  {sectors/track}
                     nhd : word;  {# heads}
                     hid : word;  {# hidden sectors}
                     end;
{Boot record}
     boot   = record jmp : array[1..3] of byte;  {jmp to boot code}
                     oem : array[1..8] of char;  {oem name}
                     inf : bpb;                  {bios parameter block}
                     cod : array[1..480] of byte;{boot code}
                     sig : word;                 {$55AA}
                     end;

{directory entry}
     dirent = record dnm : array[0..7] of char;   {name (! 0/1)}
                     ext : array[1..3] of char;   {extension}
                     att : byte;                  {attribute}
                     dos : array[12..21] of byte; {reserved area}
                     tim : word;                  {file's time}
                     dat : word;                  {file's date}
                     fcl : word;                  {first cluster number}
                     siz : longint;               {file's size}
                     end;

{file control block (default value)}
     fcb    = record drive : byte;
                     {0 = default, 1 = A,...}
                     finam : array[1..8] of char;
                     {filename & extension}
                     fiext : array[1..3] of char;
                     {left-justified with blanks}
                     cblck  :  word;
                     {current block number (0)}
                     recsz : word;
                     {record size (block = 128 rec)}
                     filsz : longint;
                     {file's size in bytes}
                     dat   : word;
                     {file's date}
                     dos   : array[0..9] of byte; {reserved area}
                     recnb :  byte;
                     {relative record nbr in cblck}
                     lgrec :  longint;
                     {logical record number (0)}
                     end;
```

```
{extended file control block}
     xfcb   = record xflag : byte;                 { $FF}
                     dos   : array[1..5] of byte; {reserved area}
                     att   : byte;                {file's attribute}
                     std   : fcb;                 {file control block}
                     end;

{search record -› bfil.(sfirst/snext) curDTA}
     srec   = record sat,                          {search attribute}
                     nx1 : byte;                   {?}
                     msk : array[1..11] of char;   {search mask 8+3}
                     nde : word;                   {dir entry number}
                     nx2 : array[1..4] of byte;    {?}
                     fcd : word;                   {first cluster of dir}
                     att : byte;                   {attribute found}
                     tim,dat : word;               {file's time & date}
                     siz : longint;                {file's size}
                     fin : array[1..13] of char;   {filename#0}
                     end;

{Primitives de manipulation des adresses disques -› int 13h}
function get_head(var dp : dsk_pos) : byte;
{    Renvoie la numéro de la tête de lecture}
function  get_sect(var dp : dsk_pos) : byte;
{    Renvoie le numéro du secteur}
function  get_cyl (var dp : dsk_pos) : word;
{    Renvoie le numéro du cylindre}
procedure def_dskp(var dp : dsk_pos; head,cylinder,sector : word);
{    Primitive de définition d'un adresse disque}


{Primitive de manipulation de date & heure dans le format -› directory}
function get_year(d : word):word;
{    Renvoie l'année (198?)}
function get_mont(d : word):byte;
{    Renvoie le numéro du mois (1..12)}
function get_day (d : word):byte;
{    Renvoie le numéro du jour dans le mois (1..31)}
function get_hour(t : word):byte;
{    Renvoie l'heure (0..23)}
function get_minu(t : word):byte;
{    Renvoie les minutes (0..59)}
function get_seco(t : word):byte;
{    Renvoie les secondes (0..59)}
function set_date(yr : word; mn,d : byte) : word;
{    Primitive de définition d'une date}
function set_time(hr,mn,sc : byte) : word;
{    Primitive de définition d'une heure}
```

```
{
Définitions relatives à une File Allocation Table (FAT) : pour des éléments
de 16 bits il suffit de propager le 12° bit des définitions ci-dessous.
}

const {rd_fat('?',0) -› media descriptor}
      med8DS = $FFF;    {FD, 2-Sided, 8 sectors}
      med8SS = $FFE;    {FD, 1-Sided, 8 sectors}
      med9DS = $FFD;    {FD, 2-Sided, 9 sectors}
      med9SS = $FFC;    {FD, 1-Sided, 9 sectors}
      medHRD = $FF8;    {Hard Disk}
      {special fat entry values}
      CluEmp = $000;    {cluster unused or available}
      CluEof = $FF8;    {last cluster of a file : ANDed test}
      CluLcf = $FFF;    {last cluster of a file}
      CluBad = $FF7;    {bad cluster : reserved}
      CluRs0 = $FF0;    {}
      CluRs1 = $FF1;    {}
      CluRs2 = $FF2;    {}
      CluRs3 = $FF3;    {Reserved cluster}
      CluRs4 = $FF4;    {}
      CluRs5 = $FF5;    {}
      CluRs6 = $FF6;    {}

{ms-dos signal values}
      signat = $AA55;   {(master) boot record signature}
      dosys  = $01;     {dos 2 system indicator}
      FAT16  = $04;     {dos system indicator with 16 bits fat entries}

function log_rd(dr : char; var buffer; debut,nombre : word) : boolean;
function log_wr(dr : char; var buffer; debut,nombre : word) : boolean;
         {cfr int 25/26h :
              dr = drive, debut = premier secteur, nombre = # sect
         }


IMPLEMENTATION {xfil.pas}
```

ANNEXE XXII. Primitives orientées "processus"

22.1. Présentation générale

Ce module permet d'accéder aux quelques facilités qu'offre le système en matière de gestion de processus : création d'un processus fils et pour celui-ci, la possibilité de retrouver ses racines (localisation du fichier de base, environnement).
En plus de cela, ce module offre quelques primitives de haut niveau qui n'existent pas explicitement dans les appels système bien qu'elles en soient des cas particuliers.

22.2. Interface de HFIL

```
unit hfil;
INTERFACE
uses bdos,xfil,mix,bfil;
const MIN_DOS : word = $1000;
      {nombre minimum de paragraphes de mémoire -› test DOSMEM}
      def_env : word = 0;
      {segment du string descripteur de l'environnement 0 => valeur du PSP}
      srch_bf : ^string = nil;{256 bytes buffer}
      {doit être défini au moins une fois avant de pouvoir utiliser Shell
       ou Exec ou tout autre primitive de recherche dans l'environnement}

type  parc    =  string[$80];{String-›paramètres terminé par #$0D ou vide}
      ppar    = ^parc;       {utiliser mix.Astr pour les constantes}

function DosMem : boolean;
{      Renvoie la valeur true si la quantité de mémoire disponible est
       supérieure à MIN_DOS paragraphes.
}

{
Pour les primitives exec et shell, la taille du heap doit être définie
laissant au minimum 64k de mémoire libres.
 le code de sortie est formé des éléments suivants :
     Le code de sortie de la fonction $4D de l'interruption 21h
  ou ($400 or dos_error_code) si une erreur se produit avant la création du
                                  processus fils.
  (bfil.curDta & srch_bf doivent être définis)
}

function exec(cmd : ppat; p : ppar) : word;
function shell(cmd : ppar) : word;
{    Primitives de création d'un processus fils.
     cmd et p doivent se trouver à une adresse supérieure au segment
     system.prefixseg.
     cmd est la ligne des paramètres passés au nouveau processus
     p est le nom du fichier .COM ou .EXE correspondant au programme.

     Pour shell, p est implicitement défini par la valeur du symbole
     COMSPEC qui se touve dans l'environnement du processus père.
}
```

```
{Primitives d'exploration de l'environnement (srch_bf doit être défini)}

function in_envir(ps : pstr) : pointer;
function comspec : pointer;
function exepath : pointer;
{    Primitives permettant de localiser les valeurs de symboles
     particuliers dans l'environnememt défini par def_env. Le résultat est
     la référence de la valeur du symbole dans l'environnement (on peut
     donc y accéder en LECTURE UNIQUEMENT). Si le symbole n'exsiste pas
     dans l'environnement courant, la valeur nil est renvoyée.
     Rappelons que l'environnement est composé d'un série d'entrées ayant
     la forme suivante : Symbole=Valeur#0
     in_envir -› renvoie la localisation du symbole ps ('=' compris).
     comspec  -› renvoie la localisation de 'COMSPEC='
     exepath  -› renvoie la localisation du path du processus courant
                 (dos › 3 uniquement).
}

function asciizTOstr(asciiz : pointer; var s : string; maxs : byte)
         :boolean;
{    Primitive permettant la transcription d'un morceau de texte terminé
     par le caractère #0 (asciiz) en l'ajoutant à la fin du string s dont
     la longueur totale ne peut dépasser maxs.
     Elle renvoie la valeur true en cas de succès de l'opération.
}

procedure save_param(var p : parc);
{    Primitive permettant de sauver dans une zone mémoire (p) la ligne de
     paramètres du processus courant.
}

{
Primitives de gestion de fichiers (curdta & srch_bf peuvent être
indéfinis), basées sur le fichier courant bfil.curFH
}

procedure cut_file;
{    Coupe le fichier à l'endroit de la position courante}

function fappend(var buffer; nb : word):word;
{    Primitive équivalente à bfil.fwrite si ce n'est qu'il s'agit d'un
     ajout en fin de fichier.
}

procedure to_disk;
{    Primitive permettant de forcer l'écriture physique des données}

IMPLEMENTATION {hfil.pas}
```

ANNEXE XXIII. FCBs réactualisés.

23.1. Présentation générale

Ce module permet d'utiliser l'ancien mode de gestion de fichiers basé sur les fcbs qui ne se justifie que dans des cas bien précis (gestion directe de directories, services particuliers n'existant pas dans la gestion "étendue" des fichiers.
Pour faciliter l'usage de ces primitives un peu lourdes, j'ai recréé des primitives qui rappellent tant que faire se peut le profil de celles que l'on retrouve dans bfil.

23.2. Interface de FCB

```
unit fcbs;
INTERFACE
uses bdos,mix,xfil,bfil; {bfil.curDta}

type xp  = ^fcb;
     PF  = record svpt,
                  {directory courante avant Hassign}
                  fnpt    : path;
                  {path complet du fichier courant}
                  f       : xfcb;
                  {"extended file control block"}
                  recsiz  : word;
                  {taille du "logical record"}
                  filsiz  : longint;
                  {nombre de "logical records" dans le fichier}
                  stdORxt :^xp;
                  {@f or @f.std pour choisir le mode étendu ou non}
                  end;
     PPF = ^PF;
     nwf = string[13]; {nom du fichier + '.'  + extension + #0}

const {mode -> fonction d'analyse (fcbparse)}
      Sls    = $01;  {passer les blancs initiaux}
      Dfd    = $02;  {modifier la valeur par déf. du drive
                        <=> il existe un identificateur de drive}
      Dff    = $04;  {modifier le nom du fichier par défaut
                        <=> le nom de fichier existe}
      Dfx    = $08;  {modifier l'extension du nom de fichier
                        <=> l'extension existe}



var  cpf    : ppf;     {pf courant -> primitives H????}
     curFcb : pointer; {fcb courant}

{(extended) fcb interne}
procedure xFiCtBk;    {extended Fcb}
procedure FiCtBk;     {curFcb par défaut}
```

```
{primitives de bas niveau (cfr DOS)}

procedure pathTOfcb(p : ppat; var svpt : path);
{    Cette  primitive  prépare un fcb non ouvert (curFCB) en fonction de  p
     ayant sauvé le path courant.
     p : path complet (drive:\directory path\filename0)
             '?','*' permis dans la parite filename
     svpt : zone de sauvegarde du path courant.
}

{
Les primitives suivantes se basent sur curFH/curDTA et suivent les spécifi-
cations des primitives correspondantes de l'interruption 21h
}
procedure fcb_open;
{ah=$0F}
procedure fcb_create;
{ah=$16}
procedure fcb_close;
{ah=$10}
procedure fcb_delete;
{ah=$13}
procedure fcb_rename;
{ah=$17}
procedure fcb_sqRead;
{ah=$14}
procedure fcb_sqWrite;
{ah=$15}
procedure sequTOrand;
{ah=$24}
procedure fcb_Fsize;
{ah=$23}
procedure fcb_rdRead;
{ah=$21}
procedure fcb_rdWrite;
{ah=$22}
procedure fcb_rdBkRd(nbrec : word);
{ah=$27}
procedure fcb_rdBkWr(nbrec : word);
{ah=$28}
function  fcb_parse(param : path; mode : byte):byte;
{ah=$24 -> code d'errreur}

{Primitives de haut niveau basées sur cpf}

procedure Hassign(p : ppat; log_rec_siz : word);
{    Primitive  d'initialisation  de  cpf^ en fonction du path p et  de  la
     taille des enregistrements logiques.
     Cette primitive est un préalable obligatoire avant H(open/create).
     Elle sauve l'environnement courant (directory).
}
```

```
procedure Hopen;
procedure Hcreate;
procedure Hclose;
{    Primitives d'ouverture/création de fichier (open/create).
     En  plus de la fermeture du fichier,  Hclose restitue  l'environnement
     sauvé lors du Hassign.
}

procedure Hrename(n : nwf);
procedure Hdelete;
{    Primitives  dont l'effet est le même que bfil.(frename/delete)  si  ce
     n'est  qu'ici,  au  lieu  de permettre la modification  pour  un  seul
     fichier, elle devient possible pour plusieurs (?/* permis). Néanmoins,
     Hrename ne permet pas de changer les fichiers de directory.
}

procedure Hseek(rn : longint);
function  BkRead(var buffer; nr : word):boolean;
function  BkWrit(var buffer; nr : word):boolean;
{    Primitives  d'accès au contenu du fichier non plus en termes de  bytes
     mais en termes d'enregistrements logiques.
     Elles correspondent à bfil.(fseek, fread, fwrite) à la différence près
     que le résultat des fonctions indique seulement le succès du transfert
     (pas le nombre d'enregistrements).
}

IMPLEMENTATION {fcbs.asm}
```

ANNEXE XXIV. Les ressources du BIOS

24.1. Présentation générale

Sous la rubrique services de l'OS, j'ai regroupé quatre petites unités qui sont en liaison étroite avec le hardware (bdos, ubreak, ???CFG) et le système d'allocation mémoire (heap).
Ces unités permettent de profiter au maximum des ressources offertes par la configuration d'un PC.

L'unité BDOS reprend les fonctionnalités de bas niveau de l'unité DOS du turbo 4. Cependant, dans l'unité DOS, la gestion des interruptions n'est pas absolument réentrante. Pour certaines applications, cela peut occasionner de désagréables surprises au niveau du stack.
De plus Bdos offre quelques possibilités supplémentaires au niveau des appels système les plus souvent utilisés (10h, 13h, 16h).

L'unité UBREAK, permet de gérer avec une grande souplesse les interruptions 'break' (1Bh) venant directement du clavier. Cette unité peut causer de désagréables interférences quand elle est utilisée conjointement avec l'unité CRT.

L'unité Heap, permet de récupérer les erreurs d'allocation mémoire en permettant de changer à tout moment le mode de traitement de ces erreurs en changeant simplement la valeur d'une variable. Ce mécanisme n'est pas valable pour le mode retry (2).

Les unités ScrCfg et SygCfg permettent de paramétrer les unités Menus, Sygr et hard_cga en fonction de la taille des applications (nombre d'objets à créer) et en fonction du type de carte graphique.

24.2. Interface de BDOS

```
unit bdos;
INTERFACE

{définition des registres du processeur 8088/8086}
type regpack = record
                  case integer of
                       0 : (ax,bx,cx,dx,bp,si,di,ds,es,fl : word);
                       1 : (x1,x2,x3,x4,x5,x6,x7,x8,x9:word;carry: boolean);
                       2 : (al,ah,bl,bh,cl,ch,dl,dh : byte);
                       end;
     minpack = record
                  case integer of
                       0 : (ax,bx,cx,dx,es : word);
                       1 : (al,ah,bl,bh,cl,ch,dl,dh:byte);
                       end;

{batterie de registre à usage général}
var rp : regpack;  mp   : minpack absolute rp;
```

```
const {zone de données -› gestion des erreurs DOS}
      xcod  : word = 0; {code d'erreur étendu    }
      ract  : byte = 0; {action proposée         }
      rclas : byte = 0; {classe de l'erreur      }
      rloc  : byte = 0; {localisation de l'erreur}
var   racl  : word absolute ract;
procedure xdosr;
{     Lorsqu'une erreur est détectée, cette primitive permet de définir les
      données ci-dessus.
}

procedure screen(var r : minpack);            {int 10}

function disk(var r : regpack):word;          {int 13}
{     Le  résultat  de cette fonction est le code d'erreur en  cas  d'échec
      (cfr flag carry) en cas de succès le résultat est nul.
}

function kbd (fnum : byte) : word;            {int 16}
{     Primitive  reprenant  les fonctionnalités de l'interruption 16h  avec
      deux  différences.  La première concerne le status du clavier où  ici
      tout le mot se trouvant à l'adresse (0:417h) est renvoyé.  La seconde
      concerne le test du buffer d'entrée,  si celui-ci est vide,  la fonc-
      tion  null  (0300h)  est renvoyée sinon c'est  le  premier  caractère
      disponible qui est renvoyé.
}

procedure msdos (var r : regpack);            {int 21}
{     Primitive d'appel de l'interruption 21h.  Au retour,  c'est la  seule
      primitive pour laquelle le booleen 'carry' est défini (fl and $0001).
}

procedure intr(inum : byte; var r : regpack); {int ??}
{     Primitive de génération d'un interruption quelconque}

{macros inline de manipulation de la représentation des registres}
procedure varES_DI(p : pointer);
procedure varDS_SI(p : pointer);
procedure varDS_DX(p : pointer);
procedure varES_BX(p : pointer);
procedure varCX_DX(p : pointer);
procedure varDX_AX(p : pointer);
{     Garnissage d'une paire de registres avec un pointeur}
function  ptrES_DI : pointer;
function  ptrDS_SI : pointer;
function  ptrDS_DX : pointer;
function  ptrES_BX : pointer;
function  ptrCX_DX : pointer;
function  ptrDX_AX : pointer;
{     Conversion d'une paire de registres en un pointeur}
function  ptrAX_00 : pointer;
function  ptrBX_00 : pointer;
{     Conversion d'un registre en un pointeur}

IMPLEMENTATION {bdos.pas, bdos.asm}
```

24.3. Interface de UBREAK

```
unit Ubreak;
INTERFACE
uses bdos; {bdos.msdos}

var  SaveInt1B : pointer;

procedure On_Break (FarProc : pointer);
{     Primitive de définition de la routine de traitement d'un break détec-
      té par BrkChk. Par défaut, une routine ne faisant rien y est assignée

      FarProc : référence d'une procédure far sans argument.
}

function  Break : boolean;
{     Primitive  de test permettant de savoir si l'utilisateur a généré  un
      break
}

procedure BreakSw (On : boolean);
{     Primitive  permettant ou non la gestion du break de l'utilisateur par
      l'unité Ubreak (modifie le vecteur de l'interruption 1Bh)
}

procedure BrkChk;
{     Cette  primitive permet de tester si l'utilisateur a généré un  break
      et  dans  l'affirmative elle effectue un appel à la routine de  trai-
      tement. (moyen le plus rapide puisque macro inline).
}

procedure ResetBrkF;
{     Remet  à zéro l'indicateur interne de l'occurence d'un  break  utili-
      sateur.
}
IMPLEMENTATION {Ubreak.pas, Ubreak.asm}
```

24.4. Interface de Heap

```
unit heap;
INTERFACE
const hrc : integer = 1; {erreur d'allocation -› nil}
IMPLEMENTATION {heap.pas}
```

24.5. Interfaces de ScrCfg et SygCfg

```
unit ScrCfg;
INTERFACE
type  VIM = (UNDEF,TX,GR); {text 80*25 or graphic 640*200}
const {hard_cga parameters}
      Video_Mem : word = $B800;  {segment memoire vidéo}
      ScreenWd  : byte = 80;     {80 bytes de 8 bits  (640 pixels}
      ScreenDp  : word = 200;    {200 lignes de 640 pixels}
      ScreenMd  : word = $2000;  {offset -› lignes impaires}
      RESET_VIM : pointer = nil; {@procédure d'initialisation vidéo}
      Def_vim   : vim  = UNDEF;  {mode par défaut -› hard_cga}

IMPLEMENTATION {ScrCfg.pas}

unit SygCfg;
INTERFACE
const mxFen : byte = 64; {nombre max de fenêtres          }
      mxFev : byte = 16; {           de fenêtres virtuelles}
      mxEcr : byte = 4;  {           d'écrans              }
      mxVue : byte = 8;  {           de vues               }
      mxCan : byte = 92; {           de canaux d'accès     }
      mxMnu : byte = 16; {           de menus              }

IMPLEMENTATION {SygCfg.pas}
```

24.6. Modification de ???Cfg

Pour modifier ???Cfg, il suffit de définir une autre unité utilisant ???Cfg qui lors de son initialisation modifiera les valeurs des constantes typées. Pour que l'ensemble des unités utilisant ???Cfg fonctionnent correctement, il faut que la déclaration de leur utilisation soit précédée par la déclaration de l'unité modifiant ???Cfg.
Par exemple : uses ....,???Cfg,modif_???Cfg,Sygr,Menu,.....

ANNEXE XXV. Modules utilitaires

25.1. Présentation générale

Les unités ci-dessous offrent un certain nombre d'extensions qui n'existent pas dans la version 4 du turbo pascal. Elles ont été regroupées en quatre petits modules :

- MIX permet la manipulation d'adresses et de strings
- XSET permet la manipulation de set dont la définition serait parallèle à la suivante : set of élément avec élément = 0..?. On se base uniquement sur l'adresse du set et son encombrement en nombre de BYTES.

-X(p)STR offre des primitives de gestion de buffers préfixés par le nombre maximum d'éléments qu'il peut contenir.

- STRW est une transposition de la gestion des string (char) pour des éléments de type word.

25.2. Interface de MIX

```
unit mix;
INTERFACE
{uses rien d'autre que system}
{type permettant la manipulation d'adresses}
const lvar  = 4;
      llong = 4;
      lstrg = 4;
      lword = 2;
      lbyte = 2;
      lchar = 2;

type  osg   = record o,s : word end;
      lh    = record l,h : byte end;

procedure CALL_FAR(farptr : pointer);
procedure CALL_NEAR(ipval : word);
{     Primitives d'appel d'une procédure sans paramètres}

function NEAR_PAR(bpofs : word) : pointer;
function FAR_PAR(bpofs : word) : pointer;
{     Accès au stack des paramètres d'une primitive}

{primitive de remplissage : cfr fillchar}
procedure fillword(var a; num,value : word);

{Primitive de gestion de 8 points cardinaux}
const Nord  = $01;
      NE    = $02;
      Est   = $04;
      SE    = $08;
      Sud   = $10;
      SO    = $20;
      Ouest = $40;
      NO    = $80;
type  shift = (d000,d045,d090,d135,d180,d225,d270,d315{,d360});
```

```
function quad (di : byte; rot : shift) : byte;
{     Macro  inline  renvoyant la direction di décalée de rot degrés   dans
      le sens des aiguilles d'une montre.
}

function Opp (di : byte) : byte;
{     Macro inline renvoyant la direction opposée (180°) à di
      Elle correspond à quad(di,d180)
}

{
Primitive de rotation de 90° d'une image composée de 8 lignes de  points  :
un byte = 8 points dont le plus à gauche est le bit de poids  le plus fort
}

procedure rot90(ori,des : pointer; npix : word);
{     ori,des sont les références des images, origine et résultat qui peu-
      vent  être   confondues.   npix  est la taille des  lignes  dont   le
      début  sera obligatoirement alligné sur un byte.
 }

{primitives de gestion de strings}
type pstr = ^string;

procedure sads(var s : string; a : string);
procedure sadc(var s : string; c : char);
{     Ajoute au string s le string a ou le caractère c}

function firoc(c : char; s : string) : byte;
function lasoc(c : char; s : string) : byte;
{     Renvoient  respectivement  la  première ou la dernière  occurence  du
      caractère c dans le string s. Si ce dernier ne s'y trouve pas ou si s
      est vide, la valeur 0 est renvoyée.
}

function nxoc(depart : byte; s : string) : byte;
{     Renvoie  la prochaine occurence du caractère se trouvant à la  posi-
      tion  depart   dans  le string s.   Si depart marquait  la   dernière
      occurence,  alors  la valeur 0 est renvoyée.  Si s est un string vide
      ou si  depart est  nul ou  dépasse la longueur de s,  alors la valeur
      0 est également renvoyée.
}

function pvoc(depart : byte; s : string) : byte;
{     Renvoie   l'occurence précédente du caractère se trouvant à la  posi-
      tion  depart   dans  le string s.   Si depart marquait  la   première
      occurence,  alors  la valeur 0 est renvoyée.  Si s est un string vide
      ou  si   depart est  <= 1 ou  dépasse la longueur  de  s,   alors  la
      valeur 0 est également renvoyée.
}
```

```
function Ostr (s : string) : word;
{     Renvoie l'offset de l'emplacement du string s.
      N'est utile que pour la version 4 du turbo pascal où tous les strings
      passés par valeur le sont en fait par référence.
      Cela permet d'accéder à des string constants qui se trouvent dans  le
      segment réservé au code de l'unité courante [Cseg:Ostr('tutu')]
}

function Astr(s : string) : pointer;
{     cfr Ostr mais ici renvoie l'addresse sous forme d'un pointeur}

IMPLEMENTATION
{mix.pas, fillword.asm, rot90.asm, posw.asm,
 nxoc.asm, pvoc.asm, firoc.asm, lasoc.asm, sads.asm, sadc.asm}
```

25.3. Interface de XSET

```
unit xset;
INTERFACE
const mxset : byte = 32;  {taille du set en bytes}
      {DOIT être initialisé AVANT TOUT appel à une des primitives
       suivantes.
      }

function  frelem(var st):integer;
{     Donne la position du premier élément absent dans l'ensemble}

function  firelem(var st):integer;
{     Donne la position du premier élément de l'ensemble
      Phase d'initialisation de nexelem
}

function  nexelem(var st):integer;
{     Donne la position de l'élément suivant de l'ensemble
      TOUTE série d'appels à cette primitive doit être précédée d'une phase
      d'initialisation (firelem)
}

function  nilelem(var st) : boolean;
{     TRUE si l'ensemble est vide}

function  tstelem(var st; elem : byte) : boolean;
{     TRUE si l'élément est présent dans l'ensemble}

procedure videset(var st);
{     Vide l'ensemble}

procedure set_on(var st; elem : byte);
{     Ajoute l'élément elem à l'ensemble}

procedure set_off(var st; elem : byte);
{     Retire l'élément elem à l'ensemble}

IMPLEMENTATION {xset.pas, geselem.asm}
```

25.4. Interface de X(p)STR

```
unit xstr; {xpstr = xstr sans passage de paramètres}
interface
type   xstring = record mxl : byte;
                        s   : string;  {taille 1..255}
                       end;
       masque  = array[0..256] of char;

{variables -› xpstr}
var    curX : pointer;         {xstring courant}
       curM : pointer;         {masque courant (alpha)}


const  {-› masques de sélection}
       skip = 0;   {ignorer le caractère}
       unic = 1;   {pas plusieurs car consécutifs}
       sepr = 2;   {séparateur entre les mots}
       unis = 3;   {unic & separ}
       blnk = 4;   {devient un blanc}
       sbgr = 8;   {symbole graphique}
       dtcr = 12;  {à ignorer lors de la conversion}

{tous les paramètres ebuf = @xstring et msk = @masque}
{pour xpstr, pour les paramètres ebuf et msk : cfr curX et curM}

procedure swp_wls(var ebuf);
{     Echange de mxl et length(s)}
procedure cut_ld(var ebuf);
{     Détruit tout caractère (‹= #$20) au début du string}
procedure cut_trl(var ebuf);
{     Détruit tout caractère (‹= #$20) en fin du string}
procedure nobrd(var ebuf);
{     Détruit tous les blancs se trouvant en fin/début de string}
procedure cut_fiw(var ebuf);
{     Détruit les mxl premiers caractères du string}
procedure siz_fiw(var ebuf; var msk);
{     mxl:=length(premier mot)}
procedure lup_fiw(var ebuf; var msk);
{     siz_fiw + upcase(premier mot)}
procedure cvt_lin(var ebuf; var msk);
{     conversion(string)}
procedure mod_lin(var ebuf; s : string; flc : char);
{     s[occurence(ebuf[i],s)]:=flc}

procedure alpha;
{masque -› separ, unic space, upcase (effectué par xlat)}

IMPLEMENTATION  {x(p)str.pas}
{cut_ld.asm, cut_trl.asm, nobrd.asm, cut_fiw.asm, siz_fiw.asm, lup_fiw.asm,
 cvt_lin.asm, mod_lin.asm, alpha.asm}

{Pcut_ld.asm,   Pcut_trl.asm,   Pnobrd.asm,   Pcut_fiw.asm,   Psiz_fiw.asm,
Plup_fiw.asm, Pcvt_lin.asm, Pmod_lin.asm, alpha.asm}
```

25.5. Interface de STRW

```
unit strw;
{module de gestion de strings de mots (word)}
INTERFACE
type stw    = string[254];
     striw  = record case byte of
                    0:(ls : byte;
                       sw : array[1..255] of word;
                      );{type normal}
                    1:(lc : byte;
                       sc : array[1..127] of word;
                      );{type compatible avec les strings}
                    2:(ss : stw;
                      );{accès en mode string -> type 1}
                    end;
     pstriw = ^striw;

{    striw[1] = array[0..(length shl 1)] of byte}

{Un mot est divisé en deux champs (cfr mix.lh) :
    le premier byte (lh.l) -> un caractère (char)
    le second  byte (lh.h) -> un code      (byte, char,...)
}

{primitives de gestion
 ATTENTION :
    - il n'y a aucun contrôle sur les bornes pour les striw comportant
      moins de 255 éléments
    - aucun contrôle de paramètres n'est effectué si ce n'est pour empêcher
      le recouvrement interdit d'une zone mémoire moyennant les restric-
      tions de contrôle sur les bornes.
    - tout paramètre incohérent donne à la primitive un comportement
      indéfini
}
{function lenw(s : pstriw):byte; macro s.ls}

function  posw(so,st : pstriw):byte;
{     (cfr  pos(so,st):byte)
      renvoie la position de so dans st sinon 0
}

procedure copw(var des : striw; s : pstriw; beg,siz : byte);
{     (cfr des:=copy(s,beg,siz))
      ajoute à des siz mots de s à partir de beg
}
procedure delw(var s : striw; beg,siz : byte);
{     (cfr del(s,beg,siz))
      détruit siz mots de s à partir de beg
}
procedure insw(so : pstriw; var sd : striw; beg : byte);
{     (cfr ins(so,sd,beg))
      insère so dans sd à la position beg
}
```

```
procedure sadsw(var s1 : striw; s2 : pstriw);
procedure sadw(var s : striw; w : word);
function  firw(w : word; s : pstriw):byte;
function  lasw(w : word; s : pstriw):byte;
function  nxw(beg : byte; s : pstriw):byte;
function  pvw(beg : byte; s : pstriw):byte;
{     cfr mix.(sads,sadc,firoc,lasoc,nxoc,pvoc)}

function  posc(c : char; s : pstriw):word;
{     renvoie la position de c dans s
      Le mode de recherche est influencé par la partie haute
      des éléments de s qui sont composés comme suit [code][char] :
          [??][0..255] : comparaison sans aucun traitement
          [05][0..127] : ignorer majuscule/minuscule  (char or  $20)
          [09][0..127] : ignorer maj/min et ctrl      (char and $1F)
      posc renvoie [xx][position] avec xx identique à ce qui existe dans s
           ou 0 si c ne se trouve pas dans s.
}

function  copsw(s : stw; beg,siz : byte) : stw;
procedure delsw(var s : stw; beg,siz : byte);
procedure inssw(so : stw; var sd : stw; beg : byte);
{     primitives de gestion en mode string
      copy, delete, insert avec beg et siz => position et nombre de MOTS
      VALABLES uniquement -> stw
      ATTENTION : length(stw) est ici le nombre de BYTES
}

IMPLEMENTATION
{strw.pas,  posw.asm,  posc.asm,  copw.asm,  delw.asm, insw.asm, sadsw.asm,
sadw.asm, firw.asm, lasw.asm, nxw.asm, pvw.asm}
```